



Open Source Computer Vision Library

Reference Manual

Copyright © 1999-2000 Intel Corporation

All Rights Reserved

Issued in U.S.A.

Order Number: TBD

World Wide Web: <http://developer.intel.com>

Version	Version History	Date
-001	Original Issue	October 24, 2000

This Open Source Computer Vision Library Reference Manual as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Open Source Computer Vision Library (OpenCV) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copyright © Intel Corporation 1999.

*Third-party brands and names are the property of their respective owners.

Contents

Chapter Contents

Chapter 1	Image Functions
Image Function Reference	1
cvCreateImageHeader.....	1
cvCreateImage	2
cvReleaseImageHeader	2
cvReleaseImage	3
cvCreateImageData.....	3
cvReleaseImageData	4
cvSetImageData	4
cvSetImageCOI	5
cvSetImageROI	5
cvGetImageRawData.....	6
cvInitImageHeader	6
cvCopyImage.....	7
Pixel Access Macro Reference.....	7
Overview.....	7
CV_INIT_PIXEL_POS	10
CV_MOVE_TO	10
CV_MOVE	11
CV_MOVE_WRAP	11
CV_MOVE_PARAM	12
CV_MOVE_PARAM_WRAP	12

Chapter 2**Dynamic Data Structures**

Memory Storage Function Reference.....	1
Overview	1
cvCreateMemStorage	3
cvCreateChildMemStorage	4
cvReleaseMemStorage	4
cvClearMemStorage	5
cvSaveMemStoragePos	5
cvRestoreMemStoragePos	5
Sequence Function Reference	7
Overview	7
cvCreateSeq	11
cvSetSeqBlockSize	12
cvSeqPush	13
cvSeqPop	13
cvSeqPushFront	14
cvSeqPopFront	14
cvSeqPushMulti	15
cvSeqPopMulti	15
cvSeqInsert	16
cvSeqRemove	16
cvClearSeq	17
cvGetSeqElem	17
cvSeqElemIdx	18
cvCvtSeqToArray	19
cvMakeSeqHeaderForArray	19
Writing and reading sequences	20
Overview	20
Reference	22
cvStartAppendToSeq	22
cvStartWriteSeq	22
cvEndWriteSeq	23

cvFlushSeqWriter	23
cvStartReadSeq.....	24
cvGetSeqReaderPos	25
cvSetSeqReaderPos.....	25
Sets	26
Overview	26
cvCreateSet	29
cvSetAdd	30
cvSetRemove.....	31
cvGetSetElem.....	31
cvClearSet	32
Graph.....	32
Overview	32
Reference	36
cvCreateGraph.....	36
cvGraphAddVtx.....	36
cvGraphRemoveVtx, cvGraphRemoveVtxByPtr	37
cvGraphAddEdge, cvGraphAddEdgeByPtr	38
cvGraphRemoveEdge, cvGraphRemoveEdgeByPtr.....	39
cvFindGraphEdge, cvFindGraphEdgeByPtr	39
cvGraphVtxDegree, cvGraphVtxDegreeByPtr	40
cvClearGraph.....	41
cvGetGraphVtx	41
cvGraphVtxIdx, cvGraphEdgeIdx.....	41

Chapter 3

Contour Processing

Basic definitions.....	1
Contour representation.....	3
Contour retrieving algorithm brief	4
Polygonal approximation	6
Douglas-Peucker Approximation	8
Reference	9
cvFindContours.....	9

cvStartFindContours	11
cvFindNextContour	12
cvSubstituteContour	12
cvEndFindContours	13
cvApproxChains	13
cvStartReadChainPoints	14
cvReadChainPoint	14
cvApproxPoly, cvApproxPolyTree	15
cvDrawContours	16
Contours moments	16
cvContoursMoments	19
cvContourArea	20
cvContourSeqArea	20
cvMatchContours	21
Hierarchical representation of Contours	22
Data Structures	28
cvCreateContourTree	29
cvContourFromContourTree	29
cvMatchContourTrees	30

Chapter 4

OpenCV Geometry

Overview	32
Ellipse Fitting	32
Line fitting	33
Convexity Defects	34
cvFitEllipse_32f	35
cvFitLine	36
cvProject3D	38
cvConvexHull, cvContourConvexHull	38
cvConvexHullApprox, cvContourConvexHullApprox	39
cvCheckContourConvexity	41
cvConvexityDefects	42
cvMinAreaRect	42

cvCalcPGH	43
cvMinEnclosingCircle.....	44

Chapter 5**OpenCV Features**

Fixed filters	1
Overview.....	1
Floating point, optimal filter kernels	6
First derivatives	6
Second derivatives.....	7
Laplacian approximation	7
Reference	8
OpenCVaplace.....	8
cvSobel	8
Feature detection functions	9
Overview	9
Corner Detection.....	9
Canny Edge Detector.....	10
Reference	13
cvCanny	13
cvPreCornerDetect	13
cvCornerEigenValsandVecs.....	14
cvCornerMinEigenVal	15
cvFindCornerSubPix.....	16
cvGoodFeaturesToTrack	17
Hough Transform	18
Overview	18
Reference	20
cvHoughLines	20
cvHoughLinesSDiv	20
cvHoughLinesP	21

Chapter 6**OpenCV Image Statistics**

Reference	1
-----------------	---

cvCountNonZero.....	1
cvSumPixels	1
cvMean	2
cvMeanMask.....	3
cvMean_StdDev.....	3
cvMean_StdDevMask	4
cvMinMaxLoc.....	4
cvMinMaxLocMask	5
cvNorm	6
cvNormMask.....	7
cvMoments	8
cvGetSpatialMoment, cvGetCentralMoment, cvGetNormalizedCentralMoment	9
cvGetHuMoments	10

Chapter 7**OpenCV Pyramids**

Overview.....	1
Reference	6
cvPyrDown.....	6
cvPyrUp	6
cvPyrSegmentation.....	7

Chapter 8**OpenCV Morphology**

Overview.....	1
Flat structuring elements for gray scale.....	3
Reference	6
cvCreateStructuringElementEx.....	6
cvReleaseStructuringElement	7
cvErode.....	7
cvDilate	8
cvMorphologyEx	9

Chapter 9	OpenCV Background Differencing
Reference	1
cvAcc	1
cvAccMask	1
cvSquareAcc	2
cvSquareAccMask	2
cvMultiplyAcc	3
cvMultiplyAccMask	3
cvRunningAvg	4
cvRunningAvgMask	5
Chapter 10	OpenCV Distance Transform
Reference	1
cvDistTransform	1
Chapter 11	OpenCV Threshold Functions
Reference	1
cvAdaptiveThreshold	1
cvThreshold	2
Chapter 12	OpenCV Flood Fill
Reference	1
cvFloodFill, cvFloodFill8	1
Chapter 13 Calibration	OpenCV Camera
Overview	1
Reference	4
cvCalibrateCamera	4
cvCalibrateCamera_64d	5
cvFindExtrinsicCameraParams	6
cvFindExtrinsicCameraParams_64d	7
cvRodrigues	7
cvRodrigues_64d	8

	cvUndistort	9
	cvFindChessBoardCornerGuesses	9
Chapter 14	OpenCV View Morphing	
	Overview	1
	Reference	3
	cvFindFundamentalMatrix	3
	cvMakeScanlines	4
	cvPreWarpImage	5
	cvFindRuns	6
	cvDynamicCorrespondMulti	6
	cvMakeAlphaScanlines	7
	cvMorphEpilinesMulti	8
	cvPostWarpImage	9
	cvDeleteMoire	9
Chapter 15	OpenCV Motion Templates	
	Overview	1
	Motion representation and normal optical flow method	1
	Reference	8
	cvUpdateMHIByTime	8
	cvCalcMotionGradient	8
	cvCalcGlobalOrientation	9
	cvSegmentMotion	10
Chapter 16	OpenCV CamShift	
	Overview	1
	Reference	7
	cvCamShift	7
	cvMeanShift	8
Chapter 17	OpenCV Eigen Objects	
	Overview	1
	Reference	4

	cvSnakeImage, cvSnakeImageGrad.....	4
Chapter 18	OpenCV Optical Flow	
	Overview.....	1
	Lucas & Kanade technique	2
	Horn & Schunck technique.....	2
	Block matching.....	3
	Reference	3
	cvCalcOpticalFlowHS	3
	cvCalcOpticalFlowLK.....	4
	cvCalcOpticalFlowBM.....	5
	cvCalcOpticalFlowPyrLK.....	6
Chapter 19	OpenCV Estimators	
	Overview.....	1
	Definitions and Motivation.....	1
	Models	1
	Estimators	2
	Kalman filtering.	2
	Reference	4
	cvCreateKalman	4
	cvReleaseKalman.....	5
	cvKalmanUpdateByTime	5
	cvKalmanUpdateByMeasurement	6
	Condensation algorithm.....	6
	Implementation of nonlinear models.....	7
	cvCreateConDensation.....	8
	cvReleaseConDensation	8
	cvConDensInitSampleSet.....	9
	cvConDensUpdatebyTime	9
Chapter 20	OpenCV POSIT	
	Overview.....	1

Background.....	1
Camera parameters.	1
Geometric image formation.....	2
Reference	7
cvCreatePOSITObject	7
cvPOSIT	8
cvReleasePOSITObject	8

Chapter 21**OpenCV Histogram**

Overview.....	1
Reference	2
cvCreateHist	2
cvReleaseHist.....	3
cvMakeHistHeaderForArray.....	3
cvQueryHistValue_1D.....	4
cvQueryHistValue_2D.....	4
cvQueryHistValue_3D.....	5
cvQueryHistValue_nD.....	6
cvGetHistValue_1D, cvGetHistValue_2D, cvGetHistValue_3D, cvGetHistValue_nD	6
cvGetMinMaxHistValue.....	7
cvNormalizeHist.....	8
cvThreshHist.....	8
cvCompareHist	8
cvCopyHist.....	9
cvSetHistThresh	10
cvCalcHist, cvCalcHistMask	10
cvCalcBackProject.....	11
cvCalcBackProjectPatch	12
cvCalcEMD	14

Chapter 22**OpenCV Gesture Recognition**

Overview.....	1
---------------	---

Reference	4
cvFindHandRegion	4
cvFindHandRegionA	5
cvCreateHandMask	6
cvCalcImageHomography	6
cvCalcProbDensity	7
cvMaxRect	8

Chapter 23

OpenCV Matrix

Overview	1
Reference	2
cvmAlloc	2
cvmAllocArray	2
cvmFree	3
cvmFreeArray	3
cvmAdd	3
cvmSub	4
cvmScale	4
cvmDotProduct	5
cvmCrossProduct	5
cvmMul	6
cvmMulTransposed	6
cvmTransp	7
cvmInvert	7
cvmTrace	8
cvmDet	8
cvmCopy	8
cvmSetZero_32f	9
cvmSetIdentity	9
cvmMahalonobis	10
cvmSVD	10
cvmEigenVV	11
cvmPerspectiveProject	12

Chapter 24	OpenCV Eigen Objects
Reference	1
cvCalcCovarMatrixEx.....	1
cvCalcEigenObjects.....	2
cvCalcDecompCoeff	3
cvEigenDecomposite	3
cvEigenProjection	4
Chapter 25	OpenCV Embedded Hidden Markov Models
Overview	1
HMM structures	1
Reference	3
cvCreate2DHMM	3
cvRelease2DHMM	3
cvCreateObsInfo	4
cvReleaseObsInfo	4
cvImgToObs_DCT.....	5
cvUniformImgSegm	6
cvInitMixSegm	6
cvEstimateHMMStateParams	7
cvEstimateTransProb	7
cvEstimateObsProb	8
cvEViterbi.....	8
cvMixSegmL2	9
Chapter 26	OpenCV Drawing Primitives
Reference	1
cvLine, cvLineAA	1
cvRectangle	2
cvCircle	2
cvEllipse, cvEllipseAA.....	3
cvFillPoly, cvFillConvexPoly	4
cvPolyLine, cvPolyLineAA.....	5

cvInitFont	5
cvPutText	6
cvGetTextSize	7

Chapter 27 Functions

OpenCV System

Reference	1
cvLoadPrimitives	1
cvGetLibraryInfo	2

Chapter 28

OpenCV Utility

Reference	1
cvAbsDiff, cvAbsDiffS	1
cvMatchTemplate	2
cvCvtPixToPlane	4
cvCvtPlaneToPix	5
cvConvertScale	5
cvInitLineIterator	6
cvSampleLine	7
cvGetRectSubPix	8
cvbFastArctan	8
cvSqrt, cvbSqrt	9
cvInvSqrt, cvbInvSqrt	10
cvbReciprocal	10
cvbCartToPolar	11
cvbFastExp	11
cvbFastLog	12
cvRandInit	12
cvbRand	13
cvFillImage	13
cvRandSetRange	14
cvKMeans	14

Chapter 29

OpenCV Bibliography

Chapter Index

Image Functions

1

Image Function Reference

cvCreateImageHeader

Allocates, initializes, and returns the structure `IplImage`.

```
IplImage* cvCreateImageHeader( CvSize size, int depth, int channels);
```

<i>size</i>	Image size (width and height).
<i>depth</i>	Image depth.
<i>channels</i>	Number of channels.

Discussion

This function allocates, initializes and returns the structure `IplImage`. This call is a shortened form of

```
iplCreateImageHeader( channels, 0, depth,  
    channels == 1 ? "GRAY" : "RGB",  
    channels == 1 ? "GRAY" : channels == 3 ? "BGR" : "BGRA",  
    IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,  
    size.width, size.height,  
    iplCreateROI(0,0,0,size.width,size.height),0,0,0);
```

Return values

Function returns the pointer to the allocated and filled structure `IplImage`.

cvCreateImage

Creates header and allocates data.

```
IplImage* cvCreateImage( CvSize size, int depth, int channels );
```

size Image size (width and height).

depth Image depth.

channels Number of channels.

Discussion

The function creates the header and allocates data. This call is a shortened form of

```
header = cvCreateImageHeader( size, depth, channels );  
cvCreateImageData( header );
```

Return values

Pointer to the allocated and filled `IplImage`.

cvReleaseImageHeader

Releases header.

```
void cvReleaseImageHeader( IplImage** image );
```

image Release the image header.

Discussion

The function releases the header. This call is a shortened form of

```
if( image )
{
    iplDeallocate( *image,
                  IPL_IMAGE_HEADER | IPL_IMAGE_ROI );

    *image = 0;
}
```

cvReleaseImage

Releases header and image data.

```
void cvReleaseImage( IplImage** image )
    image           Release the image header.
```

Discussion

The function releases the header and image data. This call is a shortened form of

```
if( image )
{
    iplDeallocate( *image, IPL_IMAGE_ALL );
    *image = 0;
}
```

cvCreateImageData

Allocates image data.

```
void cvCreateImageData( IplImage* image );
    image           Release the image header.
```

Discussion

The function allocates the image data. This call is a shortened form of

```
if( image->depth == IPL_DEPTH_32F )
{
    iplAllocateImageFP( image, 0, 0 );
}
else
{
    iplAllocateImage( image, 0, 0 );
}
```

cvReleaseImageData

Releases image data.

```
void cvReleaseImageData( IplImage* image );
```

image Release the image header.

Discussion

The function releases the image data. This call is a shortened form of

```
iplDeallocate( image, IPL_IMAGE_DATA );
```

cvSetImageData

Sets pointer to data and step parameter to given values.

```
void cvSetImageData( IplImage* image, void* data, int step );
```

image Release the image header.

<i>data</i>	User data.
<i>step</i>	Distance between the raster lines in bytes.

Discussion

The function sets the pointer to the data and step parameters to given values.

cvSetImageCOI

Sets channel of interest to given value.

```
void cvSetImageCOI( IplImage* image, int coi );
```

<i>image</i>	Release the image header.
<i>coi</i>	Channel of interest.

Discussion

The function sets the channel of interest to a given value. If ROI is `NULL` and *coi* $\neq 0$, ROI is allocated.

cvSetImageROI

Sets image ROI to given rectangle.

```
void cvSetImageROI( IplImage* image, CvRect rect );
```

<i>image</i>	Release the image header.
<i>rect</i>	ROI rectangle.

Discussion

The function sets the image ROI to a given rectangle. If ROI is `NULL` and the value of the parameter *rect* is not equal to the whole image, ROI is allocated.

cvGetImageRawData

Fills output variables with image parameters.

```
void cvGetImageRawData( const IplImage* image, uchar** data, int* step,
                        CvSize* roi_size );
```

<i>image</i>	Release the image header.
<i>data</i>	Pointer to the top-left corner of ROI.
<i>step</i>	Full width of the raster line, equals to <i>image->widthStep</i> .
<i>roi_size</i>	ROI width and height.

Discussion

The function fills the output variables with image parameters. All output parameters are optional and could be set to NULL.

cvInitImageHeader

Initializes image header structure without memory allocation.

```
void cvInitImageHeader( IplImage* image, CvSize size, int depth, int channels,
                       int origin, int align, int clear );
```

<i>image</i>	Release the image header.
<i>size</i>	Width and height of the image.
<i>depth</i>	Image depth.
<i>channels</i>	Number of channels.
<i>origin</i>	IPL_ORIGIN_TL or IPL_ORIGIN_BL.
<i>align</i>	Alignment for the raster lines.

clear If the parameter value equals to 1, the header is cleared before initialization.

Discussion

The function initializes the image header structure without memory allocation.

cvCopyImage

Copies entire image (no ROI is considered) to another.

```
void cvCopyImage(IplImage* src, IplImage* dst);
```

src Source image.

dst Destination image.

Discussion

The function copies the entire image (no ROI is considered) to another. If the destination image is smaller, the destination image data will be reallocated.

Pixel Access Macro Reference

Overview

This section describes macros that are useful for fast and flexible access to image pixels. The basic ideas behind these macros are as follow:

1. Some structures (`CvPixelAccess<type>`) are introduced. These structures contain all information about ROI and its current position. The only difference across all these structures is the data types, not the channels number.
2. There exist fast versions for moving in a specific direction, e.g., `CV_MOVE_LEFT`, the wrap and non-wrap versions. More complicated and slower macros are used for moving in an arbitrary direction (passed by a parameter).
3. Most of the macros need the parameter `cs` that specifies the number of the image channels to enable the compiler to remove superfluous multiplications in the case of the single channel, and substitute them with faster combinations in the case of three and four channels.

Example 1-1 CvPixelPosition Structures Definition

```
typedef struct _CvPixelPosition8u
{
    unsigned char*   currline;
                    /* pointer to the start of the current
                     pixel line */
    unsigned char*   topline;
                    /* pointer to the start of the top pixel
                     line */
    unsigned char*   bottomline;
                    /* pointer to the start of the first
                     line which is below the image */
    int      x;      /* current x coordinate ( in pixels ) */
    int      width;  /* width of the image ( in pixels )*/
    int      height; /* height of the image ( in pixels )*/
    int      step;   /* distance between lines ( in
                     elements of single plane ) */
    int      step_arr[3]; /* array: ( 0, -step, step ).
                           It is used for vertical
                           moving */
}
```

```
    } CvPixelPosition8u;

    /*this structure differs from the above only in data type*/
    typedef struct _CvPixelPosition8s
    {
        char*   currline;
        char*   topline;
        char*   bottomline;
        int     x;
        int     width;
        int     height;
        int     step;
        int     step_arr[3];
    } CvPixelPosition8s;

    /* this structure differs from the CvPixelPosition8u only in data type
    */
    typedef struct _CvPixelPosition32f
    {
        float*  currline;
        float*  topline;
        float*  bottomline;
        int     x;
        int     width;
        int     height;
        int     step;
        int     step_arr[3];
    } CvPixelPosition32f;
```

CV_INIT_PIXEL_POS

Initializes one of the CvPixelPosition structures.

```
#define CV_INIT_PIXEL_POS( pos, origin, step, roi, x, y, orientation )
```

<i>pos</i>	Initialized structure.
<i>origin</i>	Pointer to the left-top corner of ROI.
<i>step</i>	Width of the whole image in bytes.
<i>roi</i>	Width & height of ROI.
<i>x, y</i>	Initial position.
<i>orientation</i>	Image orientation, could be either CV_ORIGIN_TL - top/left orientation, or CV_ORIGIN_BL - bottom/left orientation.

CV_MOVE_TO

Moves to specified point (absolute shift).

```
#define CV_MOVE_TO( pos, x, y, cs )
```

<i>pos</i>	Position structure.
<i>x, y</i>	Coordinates of the new position.
<i>cs</i>	Number of the image channels.

CV_MOVE

Moves by one pixel relative to current position.

```
#define CV_MOVE_LEFT( pos, cs )
#define CV_MOVE_RIGHT( pos, cs )
#define CV_MOVE_UP( pos, cs )
#define CV_MOVE_DOWN( pos, cs )
#define CV_MOVE_LU( pos, cs )
#define CV_MOVE_RU( pos, cs )
#define CV_MOVE_LD( pos, cs )
#define CV_MOVE_RD( pos, cs )
```

pos Position structure.

cs Number of the image channels.

CV_MOVE_WRAP

*Moves by one pixel relative to current position
with wrapping when position has reached image
boundary.*

```
#define CV_MOVE_LEFT_WRAP( pos, cs )
#define CV_MOVE_RIGHT_WRAP( pos, cs )
#define CV_MOVE_UP_WRAP( pos, cs )
#define CV_MOVE_DOWN_WRAP( pos, cs )
#define CV_MOVE_LU_WRAP( pos, cs )
#define CV_MOVE_RU_WRAP( pos, cs )
#define CV_MOVE_LD_WRAP( pos, cs )
#define CV_MOVE_RD_WRAP( pos, cs )
```

pos Position structure.

cs Number of the image channels.

CV_MOVE_PARAM

Moves by one pixel in specified direction.

```
#define CV_MOVE_PARAM( pos, shift, cs )
    pos          Position structure.
    cs           Number of the image channels.
    shift        Direction. The direction value could be any of the following
                CV_SHIFT_NONE,
                CV_SHIFT_LEFT,
                CV_SHIFT_RIGHT,
                CV_SHIFT_UP,
                CV_SHIFT_DOWN,
                CV_SHIFT_UL,
                CV_SHIFT_UR,
                CV_SHIFT_DL,
```

CV_MOVE_PARAM_WRAP

Moves by one pixel in specified direction with wrapping.

```
#define CV_MOVE_PARAM_WRAP( pos, shift, cs )
    pos          Position structure.
    cs           Number of the image channels.
    shift        Direction. The direction value could be any of the following
```

CV_SHIFT_NONE,
CV_SHIFT_LEFT,
CV_SHIFT_RIGHT,
CV_SHIFT_UP,
CV_SHIFT_DOWN,
CV_SHIFT_UL,
CV_SHIFT_UR,
CV_SHIFT_DL,

Memory Storage and Sequence Reference

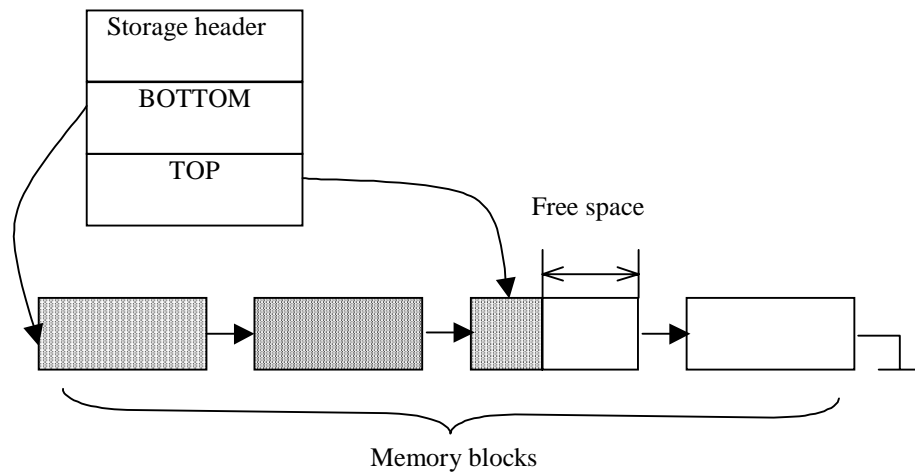
2

This chapter describes several resizable data structures and basic functions that are designed to operate on these structures.

Memory Storage Function Reference

Overview

Memory storages provide the space for storing all the dynamic data structures described in this chapter. A storage consists of a header and a double-linked list of memory blocks. This list is treated as a stack, i.e., the storage header contains a pointer to the block that is not occupied entirely and an integer value, the number of free bytes in this block. When the free space in the block has run out, the pointer is moved to the next block, if any, otherwise, a new block is allocated and then added to the list of blocks. All the blocks are of the same size and, therefore, this technique ensures an accurate memory allocation and helps avoid memory fragmentation if the blocks are rather large (see the diagram below).



Example 2-1 CvMemStorage Structure Definition

```
typedef struct CvMemStorage
{
    CvMemBlock* bottom; /* first allocated block */
    CvMemBlock* top; /* current memory block - top of the stack */
    struct CvMemStorage* parent; /* borrows new blocks from */
    int    block_size; /* block size */
    int    free_space; /* free space in the current block */
} CvMemStorage;
```

Example 2-2 CvMemBlock Structure Definition

```
typedef struct CvMemBlock
{
    struct CvMemBlock* prev;
    struct CvMemBlock* next;
} CvMemBlock;
```

Actual data of the memory blocks follows the header, i.e., the i^{th} byte of the memory block can be retrieved with the expression `((char*)(mem_block_ptr + 1))[i]`. However, the occasions on which the need for direct access to the memory blocks arises are quite rare. The structure described below stores the position of the stack top that can be saved/restored:

Example 2-3 CvMemStoragePos Structure Definition

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free_space;
}
CvMemStoragePos;
```

cvCreateMemStorage

Creates memory storage.

```
CvMemStorage* cvCreateMemStorage( int block_size );
```

block_size Size of the memory blocks in the storage (bytes).

Discussion

The function creates a memory storage and returns the pointer to it. Initially the storage is empty. All fields of the header are set to 0. The parameter *block_size* must be positive or zero; if the parameter equals to 0, the block size will be set to the default value, currently 64K.

cvCreateChildMemStorage

Creates child memory storage.

```
CvMemStorage* cvCreateChildMemStorage( CvMemStorage* parent );
```

parent Parent memory storage.

Discussion

Creates a child memory storage similar to the simple memory storage except for the differences in the memory allocation/de-allocation mechanism. When a child storage needs a new block to add to the block list, the child storage tries to get this block from the parent (first unoccupied parent block is taken and excluded from the parent block list). If no blocks are available, the parent either allocates a block or borrows one from the parent (if any). In other words, the chain (or a more complex structure) of memory storages where every storage is a child/parent of another is possible. When a child storage is released or even cleared, it returns all blocks to the parent. Note again, that in other aspects, the child storage is the same as the simple storage.

cvReleaseMemStorage

Releases memory storage.

```
void cvReleaseMemStorage( CvMemStorage** storage );
```

storage Pointer to the released storage.

Discussion

The function de-allocates all storage memory blocks or returns to the parent, if any, the header and clears the pointer to the storage. All children of the storage should be released before the parent is released.

cvClearMemStorage

Clears memory storage

```
void cvClearMemStorage( CvMemStorage* storage );
```

storage Memory storage.

Discussion

The function resets the top (free space boundary) of the storage to the very beginning. This function does not de-allocate any memory. If the storage has a parent, the function will return all blocks to the parent.

cvSaveMemStoragePos

Saves memory storage position.

```
void cvSaveMemStoragePos( CvMemStorage* storage, CvMemStoragePos* pos );
```

storage Memory storage.

pos Currently retrieved position of the in-memory storage top.

Discussion

Saves the current position of the storage top to the parameter *pos*. This position can be retrieved further by the function `cvRestoreMemStoragePos`.

cvRestoreMemStoragePos

Restores memory storage position.

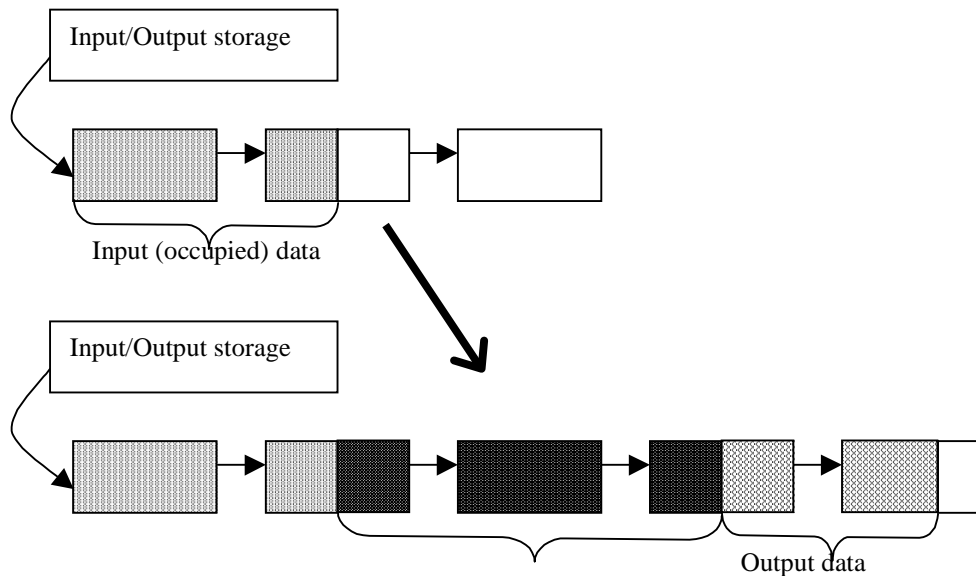
```
void cvRestoreMemStoragePos( CvMemStorage* storage, CvMemStoragePos* pos );
```

storage Memory storage.
pos New storage top position.

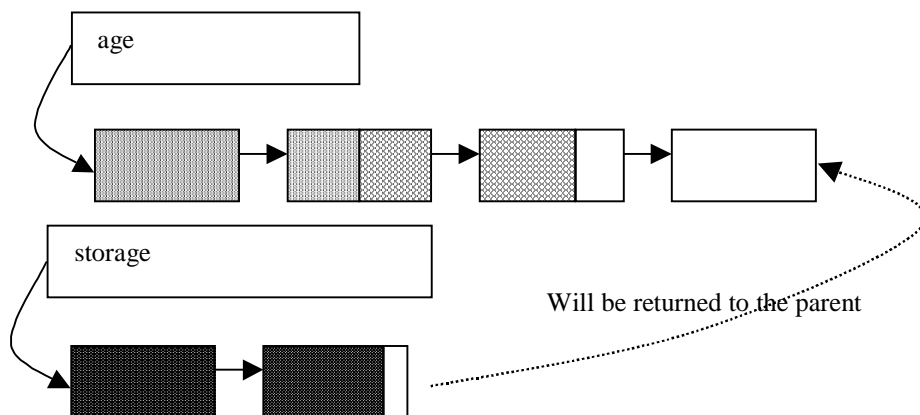
Discussion

Restores the position of the storage top from the parameter *pos*. This function and the function `cvClearMemStorage` are the only methods to release memory occupied in memory blocks.

In other words, the occupied space and free space in the storage are continuous. If the user needs to process data and put the result to the storage, there arises a need for the storage space for temporary results. In this case the user may simply write all the results to that single storage, but in this case there will be garbage in the middle of the occupied part.



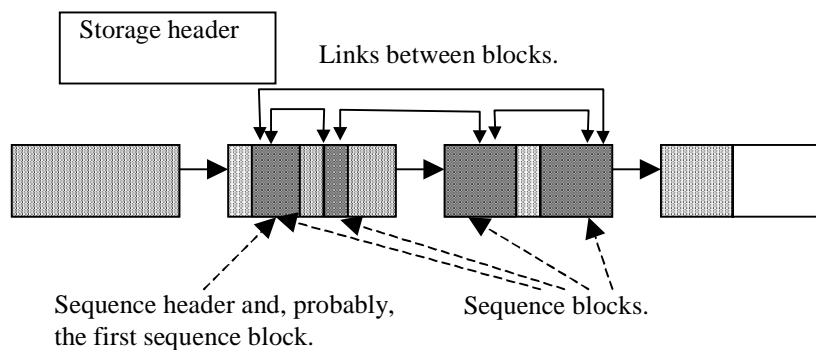
The problem is that Saving/Restoring won't work in this case. This problem, however, can be resolved by creating the child memory storage. The algorithm will write to both storages simultaneously, and, once done, it will release the temporarily storage.



Sequence Function Reference

Overview

A sequence is a resizable array of arbitrary type elements located in the memory storage. The sequence is discontinuous. The sequence data may be partitioned into several continuous blocks (called sequence blocks) that can be located in different memory blocks. The sequence blocks are connected into a circular double-linked list to store large sequences in several memory blocks or keep several small sequences in a single memory block. For example, such organization is suitable for storing contours. The sequence implementation provides fast functions for adding/removing elements to/from the head and tail of the sequence, so that the sequence implements a deque. The functions for inserting/removing elements at the middle of sequence are also available but they are slower. The sequence is the basic type for many other dynamic data structures in the library, e.g., sets, graphs, contours; just like all these types, the sequence never returns the occupied memory to the storage. However, the sequence keeps track of the memory released after removing elements from the sequence; this memory is used repeatedly. To return the memory to the storage, the user may clear a whole storage, or use save/restoring position functions, or keep temporary data in child storages.

**Example 2-4 CvSequence Structure Definition**

```

#define CV_SEQUENCE_FIELDS() \
    int      header_size; /* size of sequence header */ \
    struct   CvSeq* h_prev; /* previous sequence */ \
    struct   CvSeq* h_next; /* next sequence */ \
    struct   CvSeq* v_prev; /* 2nd previous sequence */ \
    struct   CvSeq* v_next; /* 2nd next sequence */ \
    int      flags; /* miscellaneous flags */ \
    int      total; /* total number of elements */ \
    int      elem_size; /* size of sequence element in bytes */ \
    char*    block_max; /* maximal bound of the last block */ \
    char*    ptr; /* current write pointer */ \
    int      delta_elems; /* how many elements allocated when the seq \
grows */ \
    CvMemStorage* storage; /* where the seq is stored */ \
    CvSeqBlock* free_blocks; /* free blocks list */ \
    CvSeqBlock* first; /* pointer to the first sequence block */

typedef struct CvSeq

```

```

{
    CV_SEQUENCE_FIELDS( )
} CvSeq;

```

Such an unusual definition simplifies the extension of the structure *CvSeq* with additional parameters. To extend *CvSeq* the user may define a new structure and put user-defined fields after including all *CvSeq* fields via the function *CV_SEQUENCE_FIELDS()*. The field *header_size* contains the actual size of the sequence header and must be more than or equal to *sizeof(CvSeq)*. The fields *h_prev*, *h_next*, *v_prev*, *v_next* can be used to create hierarchical structures from separate sequences. The fields *h_prev* and *h_next* point to the previous and the next sequences on the same hierarchical level while the fields *v_prev* and *v_next* point to the previous and the next sequence in the vertical direction (parent and first child). But these are just names and the pointers can be used in another fashion. The field *first* points to the first sequence block, whose structure is described below. The field *flags* contain miscellaneous information on the type of the sequence and should be discussed in more details. By convention, the lowest *CV_SEQ_ELTYPE_BITS* bits (equal to 5 in this version) contain the identifier of the element type (i.e., 32 non-overlapped element types are supported now). Predefined types are declared in the file *CVTypes.h*.

Example 2-5 Standard types of sequence elements

```

#define CV_SEQ_ELTYPE_POINT          1 /* (x,y) */
#define CV_SEQ_ELTYPE_CODE          2 /* freeman code: 0..7 */
#define CV_SEQ_ELTYPE_PPOINT        3 /* &(x,y) */
#define CV_SEQ_ELTYPE_INDEX         4 /* #(x,y) */
#define CV_SEQ_ELTYPE_GRAPH_EDGE    5 /* &next_o,&next_d,&vtx_o,
&vtx_d */
#define CV_SEQ_ELTYPE_GRAPH_VERTEX  6 /* first_edge, &(x,y) */
#define CV_SEQ_ELTYPE_TRIAN_ATR     7 /* vertex of the binary tree
*/
#define CV_SEQ_ELTYPE_CONNECTED_COMP 8 /* connected component */
#define CV_SEQ_ELTYPE_POINT3D       9 /* (x,y,z) */

```

The next *CV_SEQ_KIND_BITS* bits (also 5 bits) specify the kind of the sequence. Again, predefined kinds of sequences are declared in the file *CVTypes.h*.

Example 2-6 Standard kinds of sequences

```
#define CV_SEQ_KIND_SET          (0 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_CURVE       (1 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_BIN_TREE    (2 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
```

The remaining bits are used to identify different features specific to certain sequence kinds and element types. For example, curves made of points (`CV_SEQ_KIND_CURVE|CV_SEQ_ELTYPE_POINT`), together with the flag `CV_SEQ_FLAG_CLOSED` belong to the type `CV_SEQ_POLYGON` or its subtype, depending on other flags. Many contour processing functions check the type of the input sequence and report an error if they do not support this type. The complete list of all supported predefined sequence types and helper macros designed to get the sequence type of other properties is stored in the file `CVTypes.h`.

Below follows the definition of the building block of sequences.

Example 2-7 CvSeqBlock Structure Definition

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock*  prev; /* previous sequence block */
    struct CvSeqBlock*  next; /* next sequence block */
    int    start_index; /* index of the first element in the block +
sequence->first->start_index */
    int    count; /* number of elements in the block */
    char*  data; /* pointer to the first element of the block */
} CvSeqBlock;
```

Sequence blocks make up a circular double-linked list, so the pointers *prev* and *next* are never `NULL` and point to the previous and the next sequence blocks within the sequence (*next* of the last block is the first block and *prev* of the first block is the last block). The fields *start_index* and *count* help to track the block location within the sequence. For example, if the sequence consists of 10 elements and splits into three blocks of 3, 5, and 2 elements, and the first block has the parameter *start_index* = 2,

then pairs $\langle start_index, count \rangle$ for the sequence blocks will be $\langle 2, 3 \rangle$, $\langle 5, 5 \rangle$, and $\langle 10, 2 \rangle$ correspondingly. The parameter *start_index* of the first block is usually 0 unless some elements have been inserted in the beginning of the sequence.

cvCreateSeq

Creates sequence.

```
CvSeq* cvCreateSeq(int seq_flags, int header_size, int elem_size,
                  CvMemStorage* storage);
```

<i>seq_flags</i>	Flags of the created sequence. If the sequence will not be passed to any function working with a specific type of sequences, the sequence value may equal to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.
<i>header_size</i>	Size of the sequence header. Must be more than or equal to <i>sizeof(CvSeq)</i> . If a specific type or its extension is indicated, this type must fit the base type header.
<i>elem_size</i>	Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, if the sequence of points is created (element type <i>CV_SEQ_ELTYPE_POINT</i>), then the parameter <i>elem_size</i> must be equal to <i>sizeof(CvPoint)</i> .
<i>storage</i>	Sequence location

Discussion

The function creates a sequence and returns the pointer to it. The function allocates the sequence header in the storage block as one continuous chunk and fills the parameter *elem_size*, flags *header_size*, and *storage* with passed values, sets the parameter *delta_elems* (see the function *cvSetSeqBlockSize*) to the default value, and clears other fields, including the space behind *sizeof(CvSeq)*.



NOTE. All headers in the memory storage (including sequence headers and sequence block headers) are aligned to the 4 byte boundary.

cvSetSeqBlockSize

Sets up sequence block size.

```
void cvSetSeqBlockSize( CvSeq* seq, int block_size );
```

seq Sequence.
block_size Desirable block size.

Discussion

The function affects the memory allocation granularity. When the free space in the internal sequence buffers has run out, the function allocates *block_size* bytes in the storage. If this block follows just after the one previously allocated, the two blocks are concatenated, otherwise, a new sequence block is created. Therefore, the bigger the parameter, the lower the sequence fragmentation probability, but the more space in the storage is wasted. When the sequence is created, the parameter *block_size* is set to the default value ~1K. The function can be called any time after the sequence is created, which will affect future allocations, The final block size can be different from the desired e.g., if it is larger than the storage block size or smaller than the sequence header size plus the sequence element size.

Next four functions add or remove elements to/from one of the sequence ends. Their time complexity is $O(1)$, that is all these operations don't shift existing sequence elements

cvSeqPush

Adds element in the end of sequence.

```
void cvSeqPush( CvSeq* seq, void* element );
```

<i>seq</i>	Sequence.
<i>element</i>	Added element.

Discussion

The function pushes an element to the sequence. Although this function can be used to create a sequence element by element, there is a faster method (refer to the sequence reading and writing functions and macros below).

cvSeqPop

Removes element from the end of sequence.

```
void cvSeqPop( CvSeq* seq, void* element );
```

<i>seq</i>	Sequence.
<i>element</i>	Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

Discussion

The function removes an element from the sequence. The function reports an error if the sequence is already empty.

cvSeqPushFront

Adds element in the beginning of sequence.

```
void cvSeqPushFront( CvSeq* seq, void* element );
```

seq Sequence.

element Added element.

Discussion

The function adds an element in the beginning of the sequence.

cvSeqPopFront

Removes element from the beginning of sequence.

```
void cvSeqPopFront( CvSeq* seq, void* element );
```

seq Sequence.

element Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

Discussion

The function removes an element from the beginning of the sequence. The function reports an error if the sequence is already empty.

Next two functions are batch versions of the PUSH/POP operations.

cvSeqPushMulti

Pushes several elements to the end of sequence.

```
void cvSeqPushMulti(CvSeq* seq, int count, void* elements );
```

<i>seq</i>	Sequence.
<i>count</i>	Number of elements to PUSH.
<i>elements</i>	Pushed elements, source array.

Discussion

The function adds several elements to the end of the sequence. The added elements will be put to the sequence in the same order as they are in the input array (but they can fall into different sequence blocks).

cvSeqPopMulti

Removes several elements from the end of sequence.

```
void cvSeqPushMulti( CvSeq* seq, int count, void* elements );
```

<i>seq</i>	Sequence.
<i>count</i>	Number of elements to POP.
<i>elements</i>	Popped elements, destination array.

Discussion

The function removes several elements from the end of the sequence. If the number of the elements to be removed exceeds the total number of elements in the sequence, the function will remove as many elements as possible.

cvSeqInsert

Inserts element in the middle of sequence

```
void cvSeqInsert( CvSeq* seq, int before_index, void* element );
```

<i>seq</i>	Sequence.
<i>before_index</i>	Index before which the element is inserted. Inserting before 0 is equal to <i>cvSeqPushFront</i> and inserting before <i>seq->total</i> is equal to <i>cvSeqPush</i> . The index values in the two examples above are boundaries for allowed parameter values.
<i>element</i>	Inserted element.

Discussion

This function shifts the sequence elements from the inserted position to the nearest end of the sequence before it copies an element there. Therefore, the function is slower than the functions described above, yet the algorithm implemented is rather optimal.

cvSeqRemove

Removes element from the middle of sequence.

```
void cvSeqRemove( CvSeq* seq, int index );
```

<i>seq</i>	Sequence.
<i>index</i>	Index of removed element.

Discussion

The function removes elements by the given index. If the index is negative or greater than the total number of elements less 1, the function reports an error. An attempt to remove an element from an empty sequence is a side case of this situation. The function removes an element by shifting the sequence elements from the nearest end of the sequence *index*.

cvClearSeq

Clears the sequence.

```
void cvClearSeq( CvSeq* seq );  
    seq           Sequence.
```

Discussion

The function empties the sequence. The function does not physically return the memory to the storage, but this memory is used again when new elements are added to the sequence. This function time complexity is O(1).

cvGetSeqElem

Returns n-th element of the sequence.

```
char* cvGetSeqElem( CvSeq* seq, int index, CvSeqBlock** block );  
    seq           Sequence.  
    index         Index of element.  
    block         Optional argument. If the pointer is not NULL, the address of the  
                  sequence block that contains the element will be stored in this  
                  location.
```


Discussion

The function finds the element with the given index in the sequence and returns the pointer to it. In addition, the function can return the pointer to the sequence block that contains the element. If the element is not found, the function returns 0. The function supports negative indices where -1 stands for the last sequence element, -2 stands for the one before last, etc. If there is a big chance that the sequence consists of a single sequence block or desired element is located in the first block, then the macro `sCV_GET_SEQ_ELEM(elem_type, seq, or index)` should be used, where the parameter *elem_type* is the type of sequence elements (*CvPoint* for example), the parameter *seq* is a sequence, and the parameter *index* is the index of the desired element. The macro checks first whether the desired element belongs to the first block of the sequence and, if so, returns the element, otherwise the macro calls the main function *cvGetSeqElem*. Negative indices always cause the *cvGetSeqElem* call.

cvSeqElemIdx

Returns index of concrete sequence element.

```
int cvSeqElemIdx( CvSeq* seq, void* element, CvSeqBlock** block );
```

<i>seq</i>	Sequence.
<i>element</i>	Pointer to the element within the sequence.
<i>block</i>	Block with the found element.

Discussion

The functions returns the index of a sequence element or a negative number if the element is not found.

cvCvtSeqToArray

Copies the sequence to one continuous block of memory.

```
void* cvCvtToArray( CvSeq* seq, void* array );
```

<i>seq</i>	Sequence.
<i>array</i>	Pointer to the destination array that must fit all the sequence elements.

Discussion

The function copies the entire sequence to the specified buffer and returns the pointer to the buffer. Note, that the smart function can check whether the sequence consists of a single block an, if not, the function will allocate a temporary storage and copy the sequence to this location. The condition *seq->first->next == seq->* first checks if the sequence is continous.

cvMakeSeqHeaderForArray

Constructs sequence from array.

```
void cvMakeSeqHeaderForArray( int seq_type, int header_size, int elem_size,
    void* array, int total, CvSeq* sequence, CvSeqBlock* block );
```

<i>seq_type</i>	Type of the created sequence.
<i>header_size</i>	Size of the header of the sequence. Parameter sequence must point to the structure of that size or greater size.
<i>elem_size</i>	Size of the sequence element.
<i>array</i>	Pointer to the array that will make up the sequence.
<i>total</i>	Total number of elements in the sequence. The number of array elements must equal to the value of this parameter.

<i>sequence</i>	Pointer to the local variable that will be used as the sequence header.
<i>block</i>	Pointer to the local variable that will be the header of the single sequence block.

Discussion

The function, the exact opposite of the function `cvCvtSeqToArray`, builds a sequence from an array. The sequence will always consist of a single sequence block and the total number of elements may not be greater than the value of the parameter *total*, though the user may remove elements from the sequence, then add other elements to it with the above restriction.

Writing and reading sequences

Overview

Although the functions and macros described below are irrelevant in theory because functions like `cvSeqPush` and `cvGetSeqElem` enable the user to write to sequences and read from them, the writing/reading functions and macros are very useful in practice because of their speed.

The following problem could provide an illustrative example. We need a function to form a sequence from N random values. The PUSH version is as follows:

```
CvSeq* create_seq1( CvStorage* storage, int N ) {  
    CvSeq* seq = cvCreateSeq( 0, sizeof(*seq), sizeof(int), storage);  
    for( int i = 0; i < N; i++ ) {  
        int a = rand();  
        cvSeqPush( seq, &a );  
    }  
    return seq;  
}
```

The second version makes use of the fast writing scheme, which includes the following steps: initialization of the writing process (creating writer), writing, closing the writer (flush).

```
CvSeq* create_seq1( CvStorage* storage, int N ) {
    CvSeqWriter writer;
    cvStartWriteSeq( 0, sizeof(*seq), sizeof(int),
        storage, &writer );
    for( int i = 0; i < N; i++ ) {
        int a = rand();
        CV_WRITE_SEQ_ELEM( a, writer );
    }
    return cvEndWriteSeq( &writer );
}
```

If $N = 100000$, the first version takes 230 milliseconds to finish and the second takes 111 on Pentium III 500MHz. These characteristics assume that the storage already contains a sufficient number of blocks so that no new blocks are allocated. A comparison with the simple loop that does not use sequences will give an idea about how effective and efficient this approach is.

```
int* create_seq3( int* buffer, int N ) {
    for( i = 0; i < N; i++ ) {
        buffer[i] = rand();
    }
    return buffer;
}
```

This function takes 104 milliseconds on the same machine.

Below follow descriptions of reading/writing functions and macros. As could be seen, the sequences do not make a great impact and the difference is very insignificant (less than 7%). However, the advantage of sequences is that the user does not know in advance the amount of input or output data, these structures will enable the user to allocate memory iteratively. Another problem solution would be to use lists, yet the sequences are much faster and require less memory.

Reference

cvStartAppendToSeq

Initializes the process of writing to sequence

```
void cvAppendToSeq( CvSeq* seq, CvSeqWriter* writer );
```

<i>seq</i>	Pointer to the sequence where elements will be written.
<i>writer</i>	Pointer to the working structure that contains the current status of the writing process.

Discussion

Initializes the writer to write to the sequence. Written elements are added to the end of the sequence. Note that during the writing process other operations on the sequence may yield incorrect result or even corrupt the sequence (see also the discussion section of the function `cvFlushSeqWriter`).

cvStartWriteSeq

Creates new sequence and initializes writer for it

```
void cvStartWriteSeq(int seq_flags, int header_size, int elem_size,  
    CvMemStorage* storage, CvSeqWriter* writer);
```

<i>seq_flags</i>	Flags of the created sequence. If the sequence will not be passed to any function working with a specific type of sequences, the sequence value may equal to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.
<i>header_size</i>	Size of the sequence header. The parameter value may not be less than <code>sizeof(CvSeq)</code> . If a certain type or extension is specified, it must fit the base type header.

<i>elem_size</i>	Size of the sequence elements in bytes. Must be consistent with the sequence type. For example, if the sequence of points is created (element type <code>CV_SEQ_ELTYPE_POINT</code>), then the parameter <i>elem_size</i> must be equal to <code>sizeof(CvPoint)</code> .
<i>storage</i>	Sequence location.
<i>writer</i>	Pointer to the writer status.

Discussion

This function is the exact sum of `cvCreateSeq` and `cvStartAppendToSeq`.

cvEndWriteSeq

Finishes the writing process

```
CvSeq* cvEndWriteSeq( CvSeqWriter* writer );
```

writer Pointer to the writer status.

Discussion

This function finishes the writing process and returns the pointer to the resulting sequence. The function also truncates the last sequence block to return the whole of unfilled space to the memory storage. After that the user may read freely from the sequence and modify it.

cvFlushSeqWriter

Updates sequence headers using writer state.

```
void cvFlushSeqWriter( CvSeqWriter* writer );
```

writer Pointer to the writer status.

Discussion

The function is intended to enable the user to read sequence elements whenever required during the writing process, e.g., in order to check specific conditions. The function updates the sequence headers to make reading from the sequence possible. The writer is not closed, however, so that the writing process can be continued any time. Frequent flushes are not recommended, `cvSeqPush` is preferred.

cvStartReadSeq

Initializes process of the sequential reading from the sequence

```
void cvStartReadSeq( CvSeq* seq, CvSeqReader* reader, int reverse );
```

<i>seq</i>	Sequence.
<i>reader</i>	Pointer to the reader status.
<i>reverse</i>	Whenever the parameter value equals to 0, the reading process is going in the forward direction, i.e., from the beginning to the end, otherwise the reading process direction is reverse, from the end to the beginning.

Discussion

The function initializes the reader structure. After that all the sequence elements from the first down to the last can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM(elem, reader)` that is similar to `CV_WRITE_SEQ_ELEM`. The function puts the reading pointer to the last sequence element if the parameter *reverse* does not equal to zero. After that the macro `CV_REV_READ_SEQ_ELEM(elem, reader)` can be used to get sequence elements from the last to the first. Both macros put the sequence element to *elem* and move the reading pointer forward (`CV_READ_SEQ_ELEM`) or backward (`CV_REV_READ_SEQ_ELEM`). A circular structure of sequence blocks is used for the reading process i.e., after the last element has been read by the macro `CV_READ_SEQ_ELEM`, the first element is read when the macro is called again. The same

applies to CV_REV_READ_SEQ_ELEM. Neither function ends reading since the reading process does not modify the sequence nor requires any temporary buffers. The reader field *ptr* points to the current element of the sequence that will be read first.

cvGetSeqReaderPos

Returns index of element at the read position.

```
int cvGetSeqReaderPos( CvSeqReader* reader );
```

reader Pointer to the reader status.

Discussion

The function returns the index of the element in which the reader is currently located.

cvSetSeqReaderPos

Moves read position to specified index.

```
void cvGetSeqReaderPos( int index, int is_relative, CvSeqReader* reader );
```

index Position where the reader must be moved.

is_relative If the parameter value is not equal to zero, the index means an offset relative to the current position.

reader Pointer to the reader status.

Discussion

The function moves the read position to the absolute or relative position. This function takes into account circularity of the sequence.

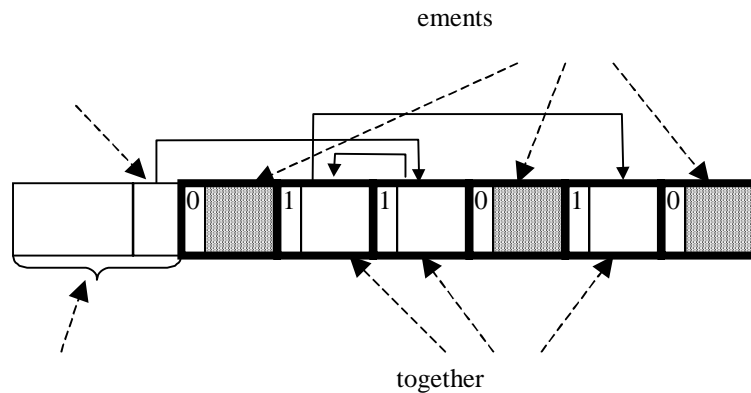
Sets

Overview

The set structure is very much based on sequences but has a totally different purpose. For example, the user will not be able to use sequences for location of the dynamic structure elements that have links between one another because if some elements have been removed from the middle of the sequence, other sequence elements will be moved to another location and their addresses and indices changed. In this case all links will have to be fixed anew. Another side of this problem is that removing elements from the middle of the sequence is slow, with time complexity of $O(n)$, where n is the number of elements in the sequence.

The problem solution lies in making the structure sparse and unordered, i.e., whenever a structure element is removed, other elements must stay where they have been and the cell previously occupied by the element be added to the pool of free cells; when a new element is inserted to the structure, the vacant cell will be used to store this new element. The set (`CvSet`) operates in this very way. The set looks like a list yet keeps no links between the structure elements. Of course, the user is free to make and keep such lists, if needed. The set is implemented as a sequence subclass; the set uses sequence elements as cells and organizes a list of free cells.

Below follows an example of a set (for simplicity, the division of the sequence/set by several memory blocks and sequence blocks is not shown).



The set elements, both existing and free cells, are all sequence elements. A special bit indicates whether the set element exists or not: on the diagram above the bits marked by 1 are free cells and the ones marked by 0 are occupied cells. The macro `CV_IS_SET_ELEM_EXISTS(set_elem_ptr)` uses this special bit to return a non zero value if the set element specified by the parameter `set_elem_ptr` belongs to set, and 0 otherwise. Below follows the definition of the structure `CvSet`:

Example 2-8 CvSet Structure Definition

```
#define CV_SET_FIELDS() \
    CV_SEQUENCE_FIELDS() \
    CvMemBlock* free_elems;

typedef struct CvSet
{
    CV_SET_FIELDS()
}
CvSet;
```

In other words a set is a sequence plus a list of free cells.

There are two modes of working with sets. The first mode uses indices for referencing the set elements within a sequence while the second mode uses pointers for the same purpose. Whereas at times the first mode is a better option, the pointer mode is faster because it does not need to find the set elements by their indices, which is done in the same way as in simple sequences. The decision on which method should be used in each particular case will depend on the type of operations to be performed on the set and the way these operations should be performed.

The ways in which a new set is created and new elements are added to the existing set are the same in either mode, the only difference between the two is in the way the elements are removed from the set. The user may even use both methods of access simultaneously, provided he or she has enough memory available to store both the index and the pointer to each element.

Like in sequences, the user may create a set with elements of arbitrary type and specify any size of the header, which, however, may not be less than `sizeof(CvSet)`. At the same time the size of the set elements is restricted to be not less than 8 bytes and divisible by 4. The reason behind this restriction is the internal set organization: if the set has a free cell available, the first 4-byte field of this set element is used as a pointer to the next free cell, which enables the user to keep track of all free cells. The second 4-byte field of the cell contains the cell to be returned when the cell becomes occupied.

When the user removes a set element when operating in the index mode, the index of the removed element is passed and stored in the released cell again. The bit indicating whether the element belongs to the set is the least significant bit of the first 4-byte field. This is the reason why all the elements must have their size divisible by 4, in this case they are all aligned to the 4-byte boundary, so that the least significant bits of their addresses are always 0.

In free cells the corresponding bit is set to 1 and, in order to get the real address of the next free cell, the functions mask this bit off. On the other hand, if the cell is occupied, the corresponding bit must equal to 0, which is the second and last restriction: the least significant bit of the first 4-byte field of the set element must be 0, otherwise the corresponding cell is considered free. If the set elements comply with this restriction, e.g., if the first field of the set element is a pointer to another set element or to some aligned structure outside the set, then the only restriction left is a non-zero number of 4- or 8-byte fields after the pointer. If the set elements do not comply with this

restriction, e.g., if the user wants to store integers in the set, the user may derive his or her structure from the structure *CvSetElem* or include it to his or her structure as the first field.

Example 2-9 CvSetElem Structure Definition

```
#define CV_SET_ELEM_FIELDS() \
    int*   aligned_ptr;
typedef struct _CvSetElem
{
    CV_SET_ELEM_FIELDS()
}
CvSetElem;
```

The first field is a dummy field and is not used in the occupied cells, except the least significant bit which is 0. With this structure the integer element could be defined as follows:

```
typedef struct _IntSetElem
{
    CV_SET_ELEM_FIELDS()
    int value;
}
IntSetElem;
```

Below follows the reference for the basic set functions.

cvCreateSet

Creates empty set.

```
CvSet* cvCreateSet( int set_flags, int header_size, int elem_size,
    CvMemStorage* storage);
```

set_flags Type of the created set.

<i>header_size</i>	Set header size. May not be less than <code>sizeof(CvSeq)</code> .
<i>elem_size</i>	Set element size. May not be less than 8 bytes, must be divisible by 4.
<i>storage</i>	Future set location.

Discussion

The function creates an empty set with the specified header size and returns the pointer to the set. The function simply redirects the call to `cvCreateSeq`.

cvSetAdd

Adds element to set.

```
int cvSetAdd( CvSet* set, CvSetElem* elem, CvSetElem** inserted_elem);
```

<i>set</i>	Set.
<i>elem</i>	Optional input argument, inserted element. If not <code>NULL</code> , the function will copy the data to the allocated cell (first 4-byte field is not copied).
<i>inserted_elem</i>	Optional output argument. Points to the allocated cell.

Discussion

The function allocates the new cell, optionally copies input element data to it, and returns the pointer and the index to the cell. The index value is taken from the second 4-byte field of the cell. In the case the cell was previously deleted and a wrong index was specified, this wrong index will be returned. However, if the user works in the pointer mode, no problem will occur and the pointer stored at *inserted_elem* may be used.

cvSetRemove

Removes element from set.

```
void cvSetRemove( CvSet* set, int index );
```

set Set.

index Index of the removed element.

Discussion

The function removes elements by their indices. This function can be used only in the index mode. If pointers are used, the macro `CV_REMOVE_SET_ELEM(set, index, elem)` may be used to take the pointer to the set, element index (any non-negative integer can be passed), and to the set element. The index of the set element could be defined by the address via the function `cvSeqElemIdx` after which the function `cvSetRemove` must be called, but this method is much slower.

cvGetSetElem

Finds set element by index.

```
CvSetElem* cvGetSetElem( CvSet* set, int index );
```

set Set.

index Index of the set element within a sequence.

Discussion

The function finds the set element by index. The function returns the pointer to it or 0 if the index is invalid or the corresponding cell is free. The function supports negative indices through calling the function `cvGetSeqElem`.



NOTE. The user can tell whether the element belongs to the set with the help of the macro `CV_IS_SET_ELEM_EXISTS(elem)` once the pointer is set to a set element.

cvClearSet

Clears set.

```
void cvClearSet( CvSet* set );
```

set Cleared set.

Discussion

The function empties the set by calling the function `cvClearSeq` and setting the pointer to the list of free cells. The function takes $O(1)$ time.

Graph

Overview

The structure set described above helps to build graphs because a graph consists of two sets, namely, vertices and edges, that refer to each other.

Example 2-10 CvGraph Structure Definition

```
#define CV_GRAPH_FIELDS() \
    CV_SET_FIELDS() \
    CvSet* edges;\ntypedef struct _CvGraph
```

```
{
    CV_GRAPH_FIELDS( )
}
CvGraph;
```

The graph structure is derived from the set of vertices and includes another set of edges. Besides, special data types exist for the graph vertices and graph edges.

Example 2-11 CvGraph Structure Definition

```
#define CV_GRAPH_EDGE_FIELDS( ) \
    struct _CvGraphEdge* next[2]; \
    struct _CvGraphVertex* vtx[2];

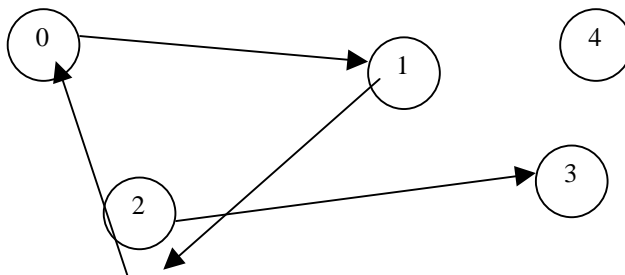
#define CV_GRAPH_VERTEX_FIELDS( ) \
    struct _CvGraphEdge* first;

typedef struct _CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS( )
}
CvGraphEdge;

typedef struct _CvGraphVertex
{
    CV_GRAPH_VERTEX_FIELDS( )
}
CvGraphVtx;
```

The graph vertex has a single predefined field that assumes the value of 1 when pointing to the first edge incident to the vertex or 0, if the vertex is isolated. The edges incident to a vertex make up the single connected non circular list. The edge structure is more complex: `vtx[0]` and `vtx[1]` are the starting and ending vertices of the edge,

`next[0]` and `next[1]` are the next edges in the incident lists for the `vtx[0]` and `vtx[1]` respectively. In other words, each edge is included in two incident lists since any edge is incident to both the starting and ending vertices. For example, consider the following oriented graph (see below for more information on non-oriented graphs).



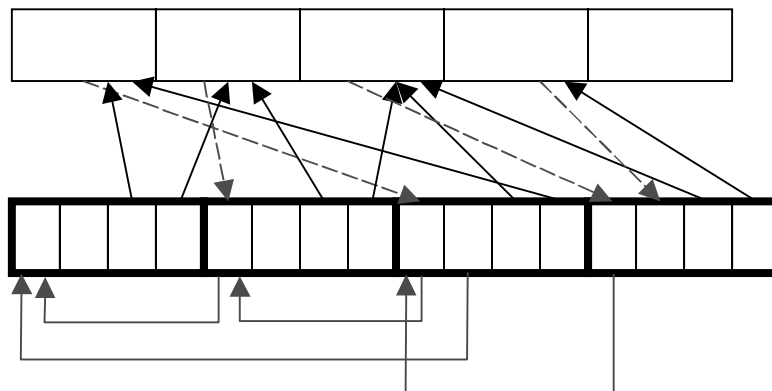
The structure can be created with the following code:

```

CvGraph* graph = cvCreateGraph( CV_SEQ_KIND_GRAPH |
CV_GRAPH_FLAG_ORIENTED,
sizeof(CvGraph),
sizeof(CvGraphVtx)+4,
sizeof(CvGraphEdge),
storage);
for( i = 0; i < 5; i++ )
{
cvGraphAddVtx( graph, 0, 0 );/* arguments like in
cvSetAdd*/
}
cvGraphAddEdge( graph, 0, 1, 0, 0 ); /* connect vertices 0
and 1, other two arguments like in cvSetAdd */
cvGraphAddEdge( graph, 1, 2, 0, 0 );
cvGraphAddEdge( graph, 2, 0, 0, 0 );
cvGraphAddEdge( graph, 2, 3, 0, 0 );

```

Internal structure will be as follows



Undirected graphs can also be represented by the structure *CvGraph*. If the oriented edges be substituted by non-oriented, the internal structure will remain the same. However, the function used to find edges will only succeed when it finds the edge from 3 to 2 because the function looks not only for edges from 3 to 2 but from 2 to 3 (and such an edge is present) as well. As follows from the code, the type of the graph is specified when the graph is created and by specifying or omitting the flag *CV_GRAPH_FLAG_ORIENTED* the user can change the behavior of the edge searching function. Two edges connecting the same vertices in undirected graphs may never be created because the existence of the edge between two vertices is checked before a new edge is inserted between them. However, internally the edge can be coded from the first vertex to the second or vice versa. Like in sets, the user may work with either indices or pointers. Note, that the graph implementation uses only pointers to refer to edges, but the user can choose indices or pointers for referencing vertices.

Reference

cvCreateGraph

Creates empty graph.

```
CvGraph* cvCreateGraph( int graph_flags, int header_size, int vertex_size, int
    edge_size, CvStorage* storage );
```

<i>graph_flags</i>	Type of the created graph. The kind of the sequence must be graph (CV_SEQ_KIND_GRAPH) and flag CV_GRAPH_FLAG_ORIENTED allows to create oriented graph. Other flags, as well as type of graph vertices and edges, are up to user.
<i>header_size</i>	Graph header size. May not be less than <i>sizeof(CvGraph)</i> .
<i>vertex_size</i>	Graph vertex size. Must be greater than <i>sizeof(CvGraphVtx)</i> to satisfy all restrictions on the set element (see above).
<i>edge_size</i>	Graph edge size. May not be less than <i>sizeof(CvGraphEdge)</i> and must be divisible by 4.
<i>storage</i>	Future location of the graph.

Discussion

The function creates an empty graph, i.e., two empty sets, a set of vertices and a set of edges, and returns it.

cvGraphAddVtx

Adds vertex to graph.

```
int cvGraphAddVtx( CvGraph* graph, CvGraphVtx* vtx, CvGraphVtx** inserted_vtx
    );
```

<i>graph</i>	Graph.
--------------	--------

<code>vtx</code>	Optional input argument. Like the parameter <code>cvSetAdd</code> , the parameter <code>vtx</code> could be used to initialize new vertices with concrete values. If <code>vtx</code> is not <code>NULL</code> , the function copies it to a new vertex, except the first 4-byte field.
<code>inserted_vtx</code>	Optional output argument. If not <code>NULL</code> , the address of the new vertex is written there.

Discussion

The function adds a vertex to the graph and returns the vertex index.

cvGraphRemoveVtx, cvGraphRemoveVtxByPtr

Remove vertex from graph.

```
void cvGraphRemoveAddVtx( CvGraph* graph, int vtx_idx );  
void cvGraphRemoveAddVtxByPtr( CvGraph* graph, CvGraphVtx* vtx );
```

<code>graph</code>	Graph.
<code>vtx_idx</code>	Index of the removed vertex.
<code>vtx</code>	Pointer to the removed vertex.

Discussion

The function removes a vertex from the graph together with all the edges incident to it. The second function puts the zero index to the appeared free cell, making further use of the vertex indeices impossible.

cvGraphAddEdge, cvGraphAddEdgeByPtr

Adds edge to graph.

```
void cvGraphAddEdge( CvGraph* graph, int start_idx, int end_idx, CvGraphEdge*
    edge, CvGraphEdge** inserted_edge );
```

```
void cvGraphAddEdge( CvGraph* graph, CvGraphVtx* start_vtx, CvGraphVtx*
    end_vtx, CvGraphEdge* edge, CvGraphEdge** inserted_edge );
```

<i>graph</i>	Graph.
<i>start_idx</i>	Index of the starting vertex of the edge.
<i>end_idx</i>	Index of the ending vertex of the edge.
<i>start_vtx</i>	Pointer to the starting vertex of the edge.
<i>end_vtx</i>	Pointer to the ending vertex of the edge.
<i>edge</i>	Optional input parameter, initialization data for the edge. If not NULL, the parameter is copied starting from 5 th 4-byte field.
<i>inserted_edge</i>	Optional output parameter that will contain the address of the inserted edge within the edge set.

Discussion

The functions add the edge to the graph given the starting and the ending vertices. The functions returns the index of the inserted edge, which is the value of the second 4-byte field of the free cell.

The functions will report an error if

- the edge that connects the vertices already exists; in this case graph orientation is taken into account;
- a pointer is NULL or indices are invalid;
- some of vertices do not exist (not checked when the pointers are passed to vertices); or
- the starting vertex is equal to the ending vertex i.e., it is impossible to create loops from a single vertex.

cvGraphRemoveEdge, cvGraphRemoveEdgeByPtr

Removes edge to graph.

```
void cvGraphRemoveEdge( CvGraph* graph, int start_idx, int end_idx );
```

```
void cvGraphRemoveEdgeByPtr( CvGraph* graph, CvGraphVtx* start_vtx,  
    CvGraphVtx* end_vtx );
```

<i>graph</i>	Graph.
<i>start_idx</i>	Index of the starting vertex of the edge.
<i>end_idx</i>	Index of the ending vertex of the edge.
<i>start_vtx</i>	Pointer to the starting vertex of the edge.
<i>end_vtx</i>	Pointer to the ending vertex of the edge.

Discussion

The functions remove the edge from the graph that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. The functions report an error if any of the vertices or edges between them do not exist.

cvFindGraphEdge, cvFindGraphEdgeByPtr

Finds edge in graph.

```
CvGraphEdge* cvFindGraphEdge( CvGraph* graph, int start_idx, int end_idx );
```

```
CvGraphEdge* cvFindGraphEdgeByPtr( CvGraph* graph, CvGraphVtx* start_vtx,  
    CvGraphVtx* end_vtx );
```

<i>graph</i>	Graph.
<i>start_idx</i>	Index of the starting vertex of the edge.
<i>end_idx</i>	Index of the ending vertex of the edge.
<i>start_vtx</i>	Pointer to the starting vertex of the edge.

end_vtx Pointer to the ending vertex of the edge.

Discussion

The functions find the graph edge that connects given vertices. If the graph is oriented, the vertices must be passed in the appropriate order. Functions return `NULL` if any of the vertices or edges between them do not exist.

cvGraphVtxDegree, cvGraphVtxDegreeByPtr

Counts edges incident to the graph vertex.

```
int cvGraphVtxDegree( CvGraph* graph, int vtx_idx );
int cvGraphVtxDegreeByPtr( CvGraph* graph, CvGraphVtx* vtx );
```

graph Graph.
vtx Pointer to the graph vertex.

Discussion

The functions count the edges incident to the graph vertex, both incoming and outgoing, and return the result. To count the edges, they use the following code:

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge ) {
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}.
```

The macro `CV_NEXT_GRAPH_EDGE(edge, vertex)` returns the next edge after the edge incident to the vertex.

cvClearGraph

Clear graph.

```
void cvClearGraph( CvGraph* graph );  
graph             Graph.
```

Discussion

The function removes all the vertices and edges from the graph. Like the function `cvClearSet`, this function takes $O(1)$ time.

cvGetGraphVtx

Finds graph vertex by index.

```
CvGraphVtx* cvGetGraphVtx( CvGraph* graph, int vtx_idx );  
graph             Graph.  
vtx_idx           Index of the vertex.
```

Discussion

The function finds the graph vertex by index and returns the pointer to it or, if not found, to a free cell at this index. Negative indices are supported.

cvGraphVtxIdx, cvGraphEdgeIdx

Returns index of graph vertex/edge.

```
int cvGraphVtxIdx( CvGraph* graph, CvGraphVtx* vtx );  
int cvGraphEdgeIdx( CvGraph* graph, CvGraphEdge* edge );
```

<i>graph</i>	Graph.
<i>vtx</i>	Pointer to the graph vertex.
<i>edge</i>	Pointer to the graph edge.

Discussion

The functions returns the index of the graph index/edge by setting pointers to them.

Contour Processing

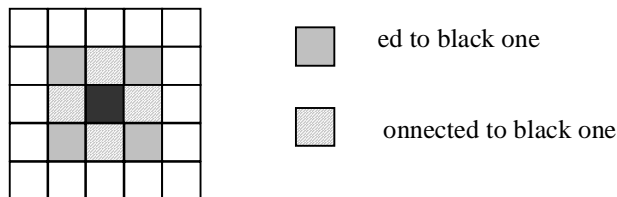
3

This chapter describes contour processing functions.

Below follow descriptions of several basic functions that scan contours on the binary image and store them in the chain format and functions for polygonal approximation of the chains.

Basic definitions

Most of the existing *vectoring* algorithms, i.e., algorithms that find contours on the raster images, deal with binary images. A binary image contains only *0-pixels* (pixels with the value 0) and *1-pixels* (pixels with the value 1). The set of *connected* 0- or 1-pixels makes the *0-(1-) component*. There are two common sorts of connectivity, the *4-connectivity* and *8-connectivity*. Two pixels with coordinates (x', y') and (x'', y'') are called 4-connected if and only if $|x' - x''| + |y' - y''| = 1$ and 8-connected if and only if $\max(|x' - x''|, |y' - y''|) = 1$. These relations are shown below:



Using this relation we can break the image into several non-overlapped 1-(0-) 4-connected (8-connected) components. Each set consists of pixels with equal values, i.e., all pixels are either equal to 1 or 0 and any pair of pixels from the set can be linked

by a sequence of 4- or 8-connected pixels. In other words, between any two points from the set a 4-(8-) path exists. The components shown on the figure below may have interrelations.



1-components W1, W2, and W3 are inside the *frame* (0-component B1), i.e., *directly* surrounded by B1.

0-components B2 and B3 are inside W1.

1-components W5 and W6 are inside B4, which is inside W3, so these 1-components are inside W3 *indirectly*. However, neither W5 nor W6 enclose one another, which means they are on the same level.

In order to avoid a topological contradiction 0-pixels must be regarded as 8-(4-) connected pixels in case 1-pixels are dealt with as 4-(8-) connected. Throughout this document we assume that 8-connectivity is used with 1-pixels and 4-connectivity with 0-pixels.

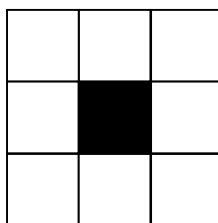
Since 0-components are complementary to 1-components, and separate 1-components are either nested to each other or their internals do not intersect, the library considers 1-components only and only their topological structure is studied, 0-pixels making up the background. A 0-component directly surrounded by a 1-component is called the

hole of the 1-component. The *border point* of a 1-component could be any pixel that belongs to the component and has a 4-connected 0-pixel. A connected set of border points is called the *border*.

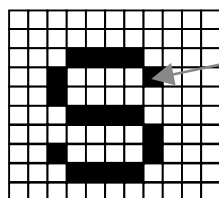
Each 1-component has a single *outer border* that separates it from the surrounding 0-component and zero or more *hole borders* that separate the 1-component from the 0-components it surrounds. It is clear that the outer border and hole borders give a full description of the component. Therefore all the borders, also referred to as *contours*, of all components stored with information about the hierarchy make up a compressed representation of the source binary image. Functions for building such a contour representation of binary images are described below.

Contour representation

The library uses two methods to represent contours. The first method is called the Freeman method or the chain code. For any pixel we can enumerate all its neighbors with numbers from 0 to 7:



Then the 0-neighbor will denote the pixel on the right side, etc. As a sequence of 8-connected points, the border can be stored as the coordinates of the initial point, followed by codes [0°7] that specify the location of the next point relative to the current one (see figure below).



e curve: 34445670007654443

The chain code is a compact representation of digital curves and an output format of the contour retrieving algorithms described below.

Polygonal representation is different option in which the curve is coded as a sequence of points, vertices of a polyline. This alternative is often a better choice for manipulating and analyzing contours over the chain codes; however, this representation is rather hard to get directly without much redundancy. Instead, algorithms that approximate the chain codes with polylines could be used.

Contour retrieving algorithm brief

Four variations of algorithms described in [1] are used in the library to retrieve borders. The first algorithm finds only the extreme outer contours on the image (the figure below shows these external boundaries of W1, W2, and W3 domains) and returns them linked to the list. The second algorithm returns all contours (total of 8 contours on the same figure below) linked to the list. The third algorithm finds all connected components by building a two-level hierarchical structure: on the top are the external boundaries of 1-domains and every external boundary contains a link to the list of holes of the corresponding component. On the figure below the external boundaries for domains W2, W5, and W6 will contain empty hole lists, the external boundary of W1 a two hole list, and W3 a list from the single hole.

The fourth algorithm returns the complete hierarchical tree where all the contours contain a list of contours surrounded by the contour directly, i.e., the hole of W3 will contain a list of two elements (W5 and W6).

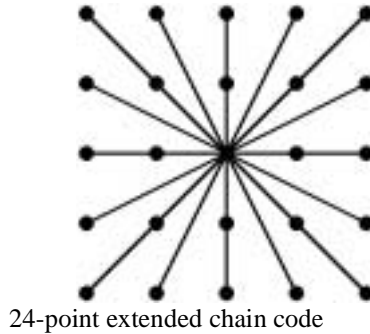


All algorithms make a single pass through the image; there are, however, rare instances when some contours need to be scanned more than once. The algorithms do raster by raster scanning.

Whenever an algorithm finds a point that belongs to a new border the border following procedure is applied to retrieve and store the border in the chain format. During the border following procedure the algorithms mark the visited pixels with special positive or negative values. If the right neighbor of the considered border point is a 0-pixel and, at the same time, the 0-pixel is located in the right hand part of the border, the border point is marked with a negative value. Otherwise, the point is marked by the same magnitude but with a positive value, if the point has not been visited yet since the border can cross itself or tangent other borders. The first and second algorithms mark all the contours with the same value and the third and fourth algorithms try to use a unique ID for each contour, which can be used to detect the parent of any newly met border.

Polygonal approximation

As soon as all the borders have been retrieved from the image, the shape representation could be further compressed. Several algorithms are available for the purpose, including RLE coding of chain codes, higher order codes (see the figure below), polygonal approximation, etc.



Polygonal approximation is the best method in terms of the output data simplicity for further processing. Below follow descriptions of two polygonal approximation algorithms. The main idea behind them is to find the dominant points i.e., points where the local curvature modulus maximums are located, on the digital curve stored in the chain code another direct representation format and leave them only. The first step here is the introduction of a discrete analog of curvature. In the continuous case curvature is determined as the speed of changing of the tangent angle:

$$k = \frac{x'y'' - x''y'}{(x'^2 + y'^2)^{3/2}}$$

In the discrete case different approximations are used. The simplest one is the difference between successive chain codes (L_1 curvature):

$$c_i^{(1)} = ((f_i - f_{i-1} + 4) \bmod 8) - 4$$

This method covers the changes from 0 (straight line) to 4 (the sharpest angle) when the direction is changed to reverse.

The following algorithm is used for getting a more complex approximation. First, for the given point (x_i, y_i) the radius m_i of the neighborhood to be considered is selected. For some algorithms m_i is a method parameter and has a constant value for all points; for others it is calculated automatically for each point. The following value is calculated for all pairs (x_{i-k}, y_{i-k}) and (x_{i+k}, y_{i+k}) ($k=1 \dots m$):

$$c_{ik} = \frac{(a_{ik} \cdot b_{ik})}{|a_{ik}| |b_{ik}|} = \cos(a_{ik}, b_{ik})$$

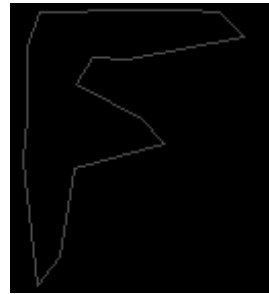
where $a_{ik} = (x_{i-k} - x_i, y_{i-k} - y_i)$, $b_{ik} = (x_{i+k} - x_i, y_{i+k} - y_i)$

The next step is finding the index h_i such that $c_{im} < c_{im-1} < \dots < c_{ih_i} \geq c_{ih_i-1}$. The value c_{ih_i} is regarded as the curvature value of the i^{th} point. The point value changes from -1 (straight line) to 1 (sharpest angle). This approximation is called the k-cosine curvature.

Rosenfeld-Johnston algorithm (RJ73) [2] is one of the earliest algorithms for determining the dominant points on the digital curves. The algorithm requires the parameter m , the neighborhood radius that often equals to $1/10$ or $1/15$ of the number of points in the input curve. Rosenfeld-Johnston algorithm is used to calculate curvature values for all points and remove points that satisfy the condition

$$\exists j, |i - j| \leq h_i/2; c_{ih_i} < c_{jh_j}$$

The remaining points are treated as dominant points. The figure below shows an example of applying the algorithm.



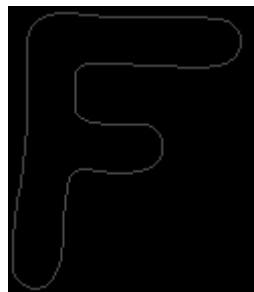
Left: source image. Right: RJ73

The disadvantage of the algorithm is the necessity of choosing the parameter m and parameter identity for all the points, which results in either excessively rough or excessively precise contour approximation.

The next algorithm proposed by Teh and Chin (TC89) [3] includes a method for the automatic selection of the parameter m for each point. The algorithm makes several passes through the curve and deletes some points at each pass. At first all points with zero $c_i^{(1)}$ curvatures are deleted (see the simplest curvature approximation). For other points the parameter m_i and the curvature value (either $c_i^{(1)}$ or k-cosine curvature) are determined. After that the algorithm performs a non-maxima suppression, same as in RJ73, deleting points whose curvature satisfies the previous condition (for $c_i^{(1)}$ the metric h_i is set to m_i). Finally, the algorithm replaces groups of two successive remaining points with a single point and groups of three or more successive points with a pair of the first and the last points. This algorithm does not require any parameters except for the curvature to use. The figure below shows the algorithm results.



Source picture



TC89 algorithm output

Douglas-Peucker Approximation

Instead of applying a quite sophisticated Teh-Chin algorithm to the chain code, the user may try another way to get a smooth contour on a little number of vertices. The idea is to apply some very simple approximation techniques to the chain code with polylines (such as substitution of horizontal, vertical, and diagonal segments by ending points) and then use the approximation algorithm on polylines. This preprocessing

reduces the amount of data without any accuracy loss. The Chin algorithm also involves this step, but uses removed points for calculating curvatures of the remaining points).

The next algorithm to consider is a pure geometrical algorithm by Douglas-Peucker for approximating a polyline with another polyline with required accuracy.

1. Two points on the given polyline are selected, thus the polyline is approximated by the line connecting these two points. The algorithm iteratively add new points to this initial approximation polyline until the required accuracy is achieved. If the polyline is not closed, two ending points are selected. Otherwise some initial algorithm should be applied to find two initial points. The more extreme the points are, the better.
2. The algorithm iterates through all polyline vertices between the two initial vertices and finds the farthest point from the line connecting two initial vertices. If this maximum distance is less than the required error, then the approximation has been found and the next segment will be taken for approximation, if any. Otherwise, the new point is added to the approximation polyline and the approximated segment is split at this point. Then two parts are approximated in the same way, since the algorithm is recursive. For a closed polygon there will be two polygonal segments to process.

Reference

cvFindContours

Finds contours in binary image.

```
void cvFindContours(IplImage* img, CvMemStorage* storage, CvSeq**  
    first_contour, int header_size, CvContourRetrievalMode mode,  
    CvChainApproxMethod method );
```

<i>img</i>	Byte-depth, single channel image. Non-zero pixels are treated as 1-pixels. The function damages the image.
<i>storage</i>	Contour storage location.

<i>first_contour</i>	Output parameter. Pointer to the first contour on the highest level.
<i>header_size</i>	Must be equal to or more than <i>sizeof(CvChain)</i> when the method <code>CV_CHAIN_CODE</code> is used and equal to or more than <i>sizeof(CvContour)</i> otherwise.
<i>mode</i>	Retrieval mode. <ul style="list-style-type: none"> • <code>CV_RETR_EXTERNAL</code> retrieves only the extreme outer contours (list); • <code>CV_RETR_LIST</code> retrieves all the contours (list); • <code>CV_RETR_CCOMP</code> retrieves the two-level hierarchy (list of connected components); • <code>CV_RETR_TREE</code> retrieves the complete hierarchy (tree).
<i>method</i>	Approximation method. <ul style="list-style-type: none"> • <code>CV_CHAIN_CODE</code> codes the output contours in the chain code; • <code>CV_CHAIN_APPROX_NONE</code> translates all the points from the chain code to points; • <code>CV_CHAIN_APPROX_SIMPLE</code> substitutes horizontal, vertical, and diagonal segments with ending points; • <code>CV_CHAIN_APPROX_TC89_L1</code>, <code>CV_CHAIN_APPROX_TC89_KCOS</code> are two versions of the Teh-Chin approximation algorithm.

Discussion

The function retrieves contours from the binary image and returns the pointer to the first contour. Other contours may be accessed through the *h_next/h_prev* (previous/next), *v_next/v_prev* (child/parent) links of the structure `CvSeq`.

cvStartFindContours

Initializes contour scanning process.

```
CvContourScanner cvStartFindContours(IplImage* img, CvMemStorage* storage, int
    header_size, CvContourRetrievalMode mode, CvChainApproxMethod method );
```

<i>img</i>	Byte-depth, single channel image. Non-zero pixels are treated as 1-pixels. The function damages the image.
<i>storage</i>	Contour storage location.
<i>first_contour</i>	Output parameter. Pointer to the first contour on the highest level.
<i>header_size</i>	Must be equal to or more than <i>sizeof(CvChain)</i> when the method <i>CV_CHAIN_CODE</i> is used and equal to or more than <i>sizeof(CvContour)</i> otherwise.
<i>mode</i>	Retrieval mode. <ul style="list-style-type: none"> • <i>CV_RETR_EXTERNAL</i> retrieves only the extreme outer contours (list); • <i>CV_RETR_LIST</i> retrieves all the contours (list); • <i>CV_RETR_CCOMP</i> retrieves the two-level hierarchy (list of connected components); • <i>CV_RETR_TREE</i> retrieves the complete hierarchy (tree).
<i>method</i>	Approximation method. <ul style="list-style-type: none"> • <i>CV_CHAIN_CODE</i> codes the output contours in the chain code; • <i>CV_CHAIN_APPROX_NONE</i> translates all the points from the chain code to points; • <i>CV_CHAIN_APPROX_SIMPLE</i> substitutes horizontal, vertical, and diagonal segments with ending points;

- CV_CHAIN_APPROX_TC89_L1, CV_CHAIN_APPROX_TC89_KCOS are two versions of the Teh-Chin approximation algorithm.

Discussion

The function initializes the contour scanner and returns the pointer to it. The structure is internal and no description is provided.

cvFindNextContour

Finds next contour on raster.

```
CvSeq* cvFindNextContour( CvContourScanner scanner );
```

scanner Contour scanner initialized by the function cvStartFindContours.

Discussion

The function returns the next contour or 0, if the image contains no more contours.

cvSubstituteContour

Replaces retrieved contour.

```
void cvSubstituteContour( CvContourScanner scanner, CvSeq* new_contour );
```

scanner Contour scanner initialized by the function cvStartFindContours.

new_contour Substituting contour.

Discussion

The function replaces the retrieved contour, that was returned from the preceding call of the function *cvFindNextContour* and stored inside the contour scanner state, with the user-specified contour. The contour will be inserted into the resulting structure, list,

two-level hierarchy, or tree, depending on the retrieval mode. If the parameter *new_contour* is 0, the retrieved contour will not be included into the resulting structure as well as all its potential children in this structure.

cvEndFindContours

Finishes scanning process.

```
CvSeq* cvEndFindContours( CvContourScanner* scanner );
```

scanner Pointer to the contour scanner.

Discussion

The function finishes the scanning process and returns the pointer to the first contour on the highest level.

cvApproxChains

Approximates Freeman chain(s) with polygonal curve.

```
void cvApproxChains( CvSeq* src_seq, CvMemStorage* storage, CvSeq** dst_seq,
    CvChainApproxMethod method, float parameter, int minimal_perimeter, int
    recursive );
```

<i>src_seq</i>	Pointer to the chain that can refer to other chains.
<i>storage</i>	Storage location for the resulting polylines.
<i>dst_seq</i>	Double pointer to the first resulting polyline.
<i>method</i>	Approximation method (see the description of the function <i>cvFindContours</i>).
<i>parameter</i>	Method parameter (not used now).

<i>minimal_perimeter</i>	Approximates only those contours whose perimeters are not less than <i>minimal_perimeter</i> . Other chains are removed from the resulting structure.
<i>recursive</i>	If not 0, approximates all chains that can be accessed from <i>src_seq</i> . If 0, approximates a single chain.

Discussion

This is a stand-alone approximation routine. The function works exactly the same way as the functions `cvFindContours/cvFindNextContour` with the corresponding approximation flag.

cvStartReadChainPoints

Initializes chain reader.

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

<i>chain</i>	Pointer to chain.
<i>reader</i>	Chain reader state.

Discussion

The function initializes the special reader (see the chapter Dynamic Data Structures for more information on sets and sequences).

cvReadChainPoint

Gets next chain point.

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

<i>reader</i>	Chain reader state.
---------------	---------------------

Discussion

The function returns the current chain point and moves to the next point.

cvApproxPoly, cvApproxPolyTree

Approximates polygonal contour(s) with desired precision.

```
void cvApproxPoly( CvSeq* src_seq, int header_size, CvMemStorage* storage,
                  CvSeq** dst_seq, CvPolyApproxMethod method, float parameter );
void cvApproxPolyTree( CvSeq* src_seq, int header_size, CvMemStorage* storage,
                      CvSeq** dst_seq, CvPolyApproxMethod method, float parameter );
```

<i>src_seq</i>	Pointer to the contour that can refer to other chains.
<i>header_size</i>	Size of the header for resulting sequences.
<i>storage</i>	Resulting contour storage location.
<i>dst_seq</i>	Double pointer to the first resulting contour.
<i>method</i>	Approximation method (only CV_POLY_APPROX_DP, Douglas-Peucker method is supported).
<i>parameter</i>	Method parameter (for CV_POLY_APPROX_DP this is a desired precision).

Discussion

The first function approximates the single contour and the second is the wrapper of the first one intended for batch processing.

cvDrawContours

Draws contours on image.

```
void cvDrawContours( IplImage *img, CvSeq* contour, int external_color, int
hole_color, int max_level );
```

<i>img</i>	Image where the contours will be drawn. As usual, all output is clipped with the ROI.
<i>contour</i>	Pointer to the first contour.
<i>external_color</i>	Color to draw external contours with.
<i>hole_color</i>	Color to draw holes with.
<i>max_level</i>	Maximal level for drawn contours. If 0, only the contour is drawn. If 1, the contour and all contours on the same level (which are after contour) are drawn. If 2, all contours after and all contours one level below the contours are drawn, etc.

Discussion

The function draws contour outlines on the image.

Contours moments

The moment of order (p; q) of an arbitrary region R is given by

$$v_{pq} = \iint_R x^p \cdot y^q dx dy \quad (7.1)$$

If $p = q = 0$, we obtain the area a of R . The moments are usually normalized by the area a of R . These moments are called normalized moments:

$$\alpha_{pq} = (1/a) \iint_R x^p \cdot y^q dx dy \quad (7.2)$$

Thus $\alpha_{00} = 1$. For $p + q \geq 2$ normalized Central moments of R are usually the ones of interest:

$$\mu_{pq} = 1/a \iint_R (x - a_{10})^p \cdot (y - a_{01})^q dx dy \quad (7.3)$$

It is an explicit method for calculation of moments of arbitrary closed polygons. Contrary to most implementations that obtain moments from the discrete pixel data, this approach calculates moments by using only the border of a region. Since no explicit region needs to be constructed, and because the border of a region usually consists of significantly fewer points than the entire region, the approach is very efficient. The well-known Green's formula is used to calculate moments.

$$\iint_R (\partial Q / (\partial x - \partial P / \partial y)) dx dy = \int_b (P dx + Q dy),$$

where b is the border of the region R .

From (7.1) we may consider

$$\partial Q / \partial x = x^p \cdot y^q, \partial P / \partial y = 0,$$

Hence

$$P(x, y) = 0, Q(x, y) = 1/(p+1) \cdot x^{p+1} y^q$$

Therefore, the moments from (7.1) can be calculated as follows.

$$\mu_{pq} = \int_b (1/(p+1) x^{p+1} \cdot y^q) dy \quad (7.4)$$

If border b consists of n points $p_i = (x_i, y_i)$, $0 \leq i \leq n$, $p_0 = p_n$, we may write

$$b(t) = \bigcup_{i=1}^n b_i(t),$$

where $b_i(t)$, $t \in [0,1]$ is set by

$$b_i(t) = t p_i + (1-t) p_{i-1}$$

Therefore, (7.4) can be calculated in the following manner

$$\mu_{pq} = \sum_{i=1}^n \int_{b_i} (1/(p+1) x^{p+1} \cdot y^q) dy \quad (7.5)$$

After unnormalized moments have been transformed, (7.5) could be written as:

$$v_{pA} = \frac{1}{(p+q+2)(p+q+1) \binom{p+q}{p}} \times \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1}) \sum_{k=0}^p \sum_{t=0}^q \binom{k+t}{t} \binom{p+q-k-t}{q-t} x_i^k x_{i-1}^{p-k} y_i^t y_{i-1}^{q-t}$$

Central unnormalized and normalized moments up to order 3 will look like

$$a = 1/2 \sum_{i=1}^n x_{i-1}y_i - x_i y_{i-1},$$

$$a_{10} = 1/(6a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(x_{i-1} + x_i),$$

$$a_{01} = 1/(6a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(y_{i-1} + y_i),$$

$$a_{20} = 1/(12a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(x_{i-1}^2 + x_{i-1}x_i + x_i^2),$$

$$a_{11} = 1/(24a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(2x_{i-1} + x_{i-1}y_i + x_i y_{i-1} + 2x_i y_i)$$

$$a_{02} = 1/(12a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(y_{i-1}^2 + y_{i-1}y_i + y_i^2)$$

$$a_{30} = 1/(20a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(x_{i-1}^3 + x_{i-1}^2 x_i + x_i^2 x_{i-1} + x_i^3),$$

$$a_{21} = 1/(60a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(x_{i-1}^2(3y_{i-1} + y_i) + 2x_{i-1}x_i(y_{i-1} + y_i) + x_i^2(y_{i-1} + 3y_i))$$

$$a_{12} = 1/(60a) \sum_{i=1}^n (x_{i-1}y_i - x_i y_{i-1})(y_{i-1}^2(3x_{i-1} + x_i) + 2y_{i-1}y_i(x_{i-1} + x_i) + y_i^2(x_{i-1} + 3x_i))$$

$$\begin{aligned}
a_{03} &= 1/(20a) \sum^n (x_{i-1}y_i - x_iy_{i-1})(y_{i-1}^3 + y_{i-1}^2y_i + y_i^2y_{i-1} + y_i^3), \\
\mu_{20} &= \alpha_{20} - \alpha_{10}^2, \\
\mu_{11} &= \alpha_{11} - \alpha_{10}\alpha_{01}, \\
\mu_{02} &= \alpha_{02} - \alpha_{01}^2, \\
\mu_{30} &= \alpha_{30} + 2\alpha_{10}^3 - 3\alpha_{10}\alpha_{20}, \\
\mu_{21} &= \alpha_{21} + 2\alpha_{10}^3\alpha_{01} - 2\alpha_{10}\alpha_{11} - \alpha_{20}\alpha_{01}, \\
\mu_{12} &= \alpha_{12} + 2\alpha_{01}^3\alpha_{10} - 2\alpha_{01}\alpha_{11} - \alpha_{02}\alpha_{10}, \\
\mu_{03} &= \alpha_{03} + 2\alpha_{01}^3 - 3\alpha_{01}\alpha_{02}
\end{aligned}$$

cvContoursMoments

Calculates contour moments up to order 3.

```
void cvContoursMoments(CvSeq* contour, CvMoments* moments);
```

<i>contour</i>	Pointer to the input contour header.
<i>moments</i>	Pointer to the output structure of contour moments (must be allocated by the caller).

Discussion

The function calculates unnormalized spatial and central moments of the contour up to order 3.

cvContourArea

Calculate the region area inside the contour.

```
void cvContourArea(CvSeq* contour, double* area);
```

contour Pointer to the input contour header.

area Pointer to the calculated area.

Discussion

The function calculates the region area within the contour consisting of n points

$p_i = (x_i, y_i)$, $0 \leq i \leq n$, $p_0 = p_n$, as a spatial moment:

$$\alpha_{00} = 1/2 \sum_{i=1}^n x_{i-1}y_i - x_i y_{i-1}$$

If the input contour has got points of self-intersection, the region area within the contour may be calculated incorrectly.

cvContourSeqArea

Calculates region area within contour section between two points.

```
void cvContourSecArea(CvSeq* contour, int n1, int n2, double* area);
```

contour Pointer to the input contour header.

n1, n2 Numbers of the two points on the contour.

area Value of the calculated area.

Discussion

The function calculates the region area within the contour section between the contour points that contain numbers from $n1$ to $n2$ and the line connecting the points $n1$ and $n2$. If the segment of the contour from the point $n1$ to the point $n2$ intersects this line, the sum of the all sections area is calculated. If the input contour has got points of self-intersection, the region area within the contour may be calculated incorrectly.

cvMatchContours

Calculates match measure between two contours.

```
void cvMatchContours (CvSeq *contour_1, CvSeq* contour_2, int method, double*
    result);
```

<i>contour_1</i>	Pointer to the first input contour header.
<i>contour_2</i>	Pointer to the second input contour header.
<i>result</i>	Pointer to the calculated value of similarity measure.
<i>method</i>	Method for the similarity measure calculation, must be any of <ul style="list-style-type: none"> • CV_CONTOURS_MATCH_I1; • CV_CONTOURS_MATCH_I2; • CV_CONTOURS_MATCH_I3.

Discussion

The function calculates one of the three similarity measures between two contours.

Let two closed contours A and B have n and m points respectively:

$A = \{(x_i, y_i), 1 \leq i \leq n\}$ $B = \{(u_i, v_i), 1 \leq i \leq m\}$. For these contours normalized central moments η_{pq} , $0 \leq p + q \leq 3$. Hu has shown that a set of next seven features derived from the second and third moments of contours is an invariant to translation, rotation, and scale change.

$$h_1 = \eta_{20} + \eta_{02}$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$h_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\ + -(\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

From these seven invariant features the three similarity measures I_1 , I_2 , and I_3 may be calculated:

$$I_1(A, B) = \sum_{i=1}^7 \left| -1/m_i^A + 1/m_i^B \right|,$$

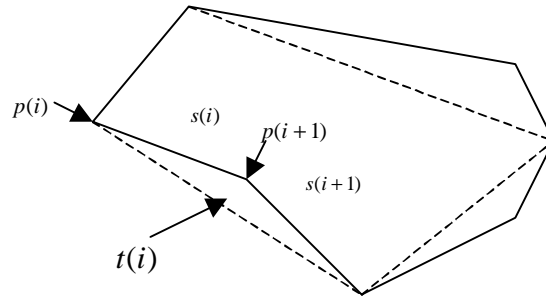
$$I_2(A, B) = \sum_{i=1}^7 \left| -m_i^A + m_i^B \right|,$$

$$I_3(A, B) = \max_i \left| (m_i^A - m_i^B) / m_i^A \right|,$$

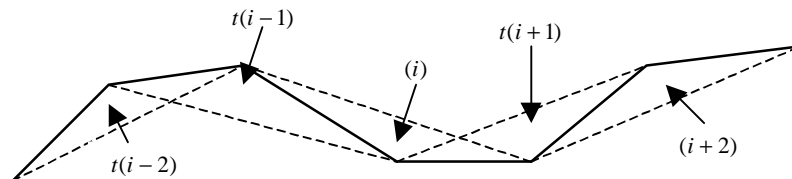
$$\text{where } m_i^A = \text{sgn}(h_i^A) \log_{10} |h_i^A| \quad . \quad m_i^B = \text{sgn}(h_i^B) \log_{10} |h_i^B| \quad .$$

Hierarchical representation of Contours

Let T be the simple closed boundary of a shape with n points $T: \{p(1), p(2), \dots, p(n)\}$ and n runs: $\{s(1), s(2), \dots, s(n)\}$. Every run $s(i)$ is formed by the two points $(p(i), p(i+1))$. For every pair of the neighboring runs $s(i)$ and $s(i+1)$ a triangle is defined by the two runs and the line connecting the two far ends of the two runs (fig.7.1).

Figure 3-1

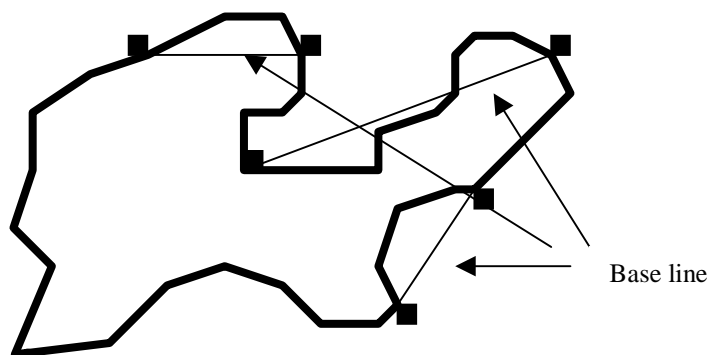
We call triangles $t(i-2)$, $t(i-1)$, $t(i+1)$, $t(i+2)$ the neighboring triangles of $t(i)$ (fig. 7.2).

Figure 3-2

For every straight line that connects any two different vertices of a shape, the line either cuts away a region from the original shape or fills in a region of the original shape or does both. The size of the region is called the interceptive area of that line (Fig. 7.3). This line we call the base line of the triangle.

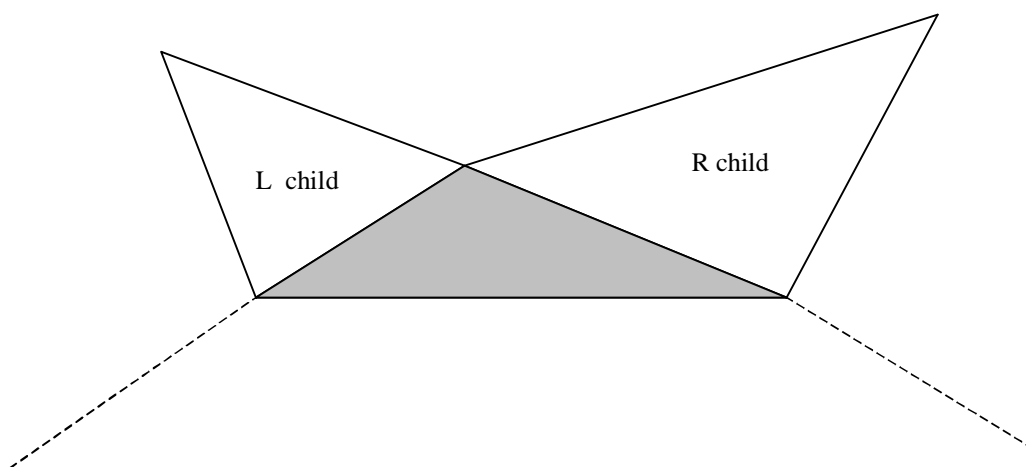
A triangle made of two boundary runs is the locally minimum interceptive area triangle (LMIAT) if the interceptive area of its base line is smaller than both its neighboring triangles areas.

Figure 3-3



The shape-partitioning algorithm is multilevel. This procedure subsequently removes some points from the contour, the removed points become children nodes of the tree. On each iteration the procedure examines the triangles defined by all the pairs of the neighboring edges along the shape boundary and finds all LMIATs. After that all LMIATs whose areas are too small (less than a reference value, which is algorithm parameter) are removed (remove middle points). If the user wants to get a precise representation, zero reference value could be passed. Other LMIATs are also removed, but the corresponding middle points are stored into the tree, after which another iteration is run. This process ends when the shape has been simplified to a quadrangle. The algorithm will then determine a diagonal line that divides this quadrangle into two triangles in the most unbalanced way.

Thus the binary tree representation is constructed from the bottom to top levels. Every tree node is associated with one triangle. Except the root node, every node is connected to its parent node, and every node may have none, or single, or two child nodes. Each newly generated node becomes the parent of the nodes for which the two sides of the new node form the base line. The triangle that uses the left side of the parent triangle is the left child. The triangle that uses the right side of the parent triangle is the right child (fig. 7.4).

Figure 3-4

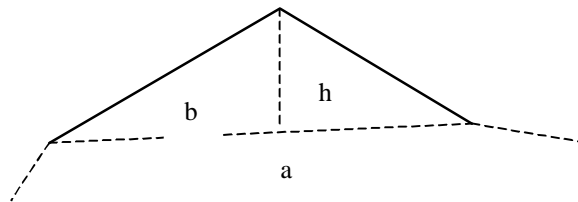
The root node associates with the diagonal line of the quadrangle. This diagonal line divides the quadrangle into two triangles. The larger triangle is the left child and the smaller triangle is its right child.

For any tree node we record the following attributes:

- Coordinates x and y of the vertex P that do not lie on the base line of LMIAT (i.e., coordinates of the middle (removed) point);
- Area of the triangle;
- Ratio of the height of the triangle h to the length of the base line a (fig. 7.5);

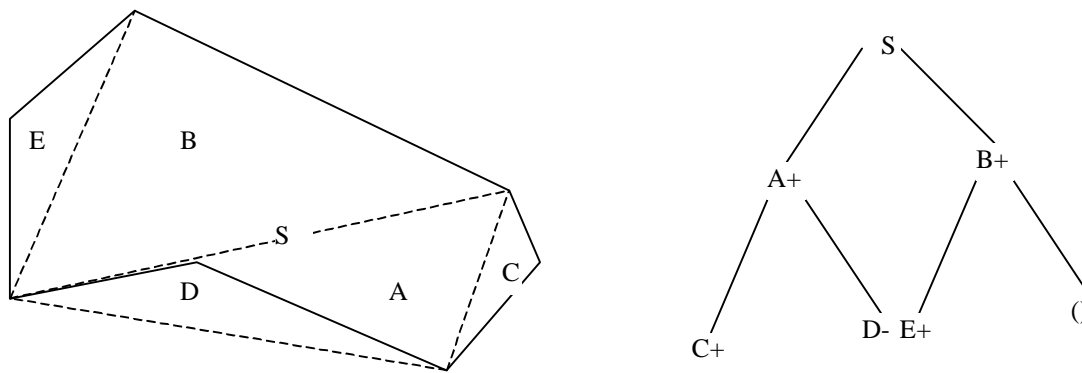
- Ratio of the projection of the left side of the triangle on the base line b to the length of the base line a ;
- Signs “+” or “-” (the sign “+” indicates that the triangle lies outside of the new shape due to the ‘cut’ type merge; the sign “-” indicates that the triangle lies inside the new shape).

Figure 3-5



The example of the shape partitioning is shown in fig. 7.6.

Figure 3-6



It is necessary to note that only the first attribute is sufficient for source contour reconstruction; all other attributes may be calculated from it. However, the other four attributes are very helpful for efficient contour matching.

The shape matching process that compares two shapes to determine whether they are similar or not can be done by matching two corresponding tree representations, e.g., two trees would be compared from top to bottom, node by node, using the breadth-first traversing procedure.

Let us define the *corresponding node pair* (CNP) of two binary tree representations TA and TB . The corresponding node pair is called $[A(i), B(i)]$, if $A(i)$ and $B(i)$ are at the same level and same position in their respective trees.

The next step is defining the *node weight*. The weight of $N(i)$ denoted as $w[N(i)]$ is defined as the ratio of the size of $N(i)$ to the size of the entire shape.

Let $N(i)$ and $N(j)$ be two nodes with heights $h(i)$ and $h(j)$ and base lengths $a(i)$ and $a(j)$ respectively. The projections of their left sides on their base lines are $b(i)$ and $b(j)$ respectively. The node distance between $N(i)$ and $N(j)$, $dn[N(i), N(j)]$, is defined as:

$$dn[N(i), N(j)] = |h(i)/a(i) \cdot w[N(i)] \mp h(j)/a(j) \cdot w[N(j)]| + |b(i)/a(i) \cdot w[N(i)] \mp b(j)/a(j) \cdot w[N(j)]|$$

In the above equation, the “+” signs are used when the sign attributes to the two nodes are different and the “-” signs are used when the two nodes have the same sign.

For two trees TA and TB representing two shapes SA and SB and with the corresponding node pairs $[A(1), B(1)], [A(2), B(2)], \dots, [A(n), B(n)]$ the tree distance between TA and TB , $dt(TA, TB)$, is defined as:

$$dt(TA, TB) = \sum_{i=1}^k dn[A(i), B(i)]$$

If the two trees are different in size, the smaller tree will be enlarged with trivial nodes so that the two trees can be fully compared. A trivial node is a node whose size attribute is zero. Thus, the trivial node weight is also zero. The values of other node attributes are trivial and not used in matching. The sum of the node distances of the first k CNPs of TA and TB is called the cumulative tree distance $dt(TA, TB, k)$ and is defined as:

$$dc(TA, TB, k) = \sum_{i=1}^k dn[A(i), B(i)]$$

Cumulative tree distance shows the dissimilarity between the approximations of the two shapes and exhibits the multiresolution nature of the tree representation in shape matching.

The shape matching algorithm is quite straightforward. For two given tree representations we traverse the two trees according to the breadth-first sequence to find CNPs of the two trees. Next we calculate for every i $dn[A(i), B(i)]$ and $dc(TA, TB, i)$. If for some i $dc(TA, TB, i)$ is larger than the tolerance threshold value, the matching procedure is terminated to indicate that the two shapes are dissimilar, otherwise continue. If $dt(TA, TB)$ is still less than the tolerance threshold value, then the procedure is terminated to indicate that there is a good match between TA and TB .

Data Structures

The Computer Vision Library functions use special data structures to represent the contours and contour binary tree in memory, namely the structures `CvSeq` and `CvContourTree`. Below follows the definition of the structure `CvContourTree` in the C language.

Example 3-1 `CvContourTree` Structure Definition

```
typedef struct CvContourTree
{
    CV_SEQUENCE_FIELDS(
        CvPoint p1;           /*the start point of the binary tree
                               root*/
        CvPoint p2;           /*the end point of the binary tree
                               root*/
    )
} CvContourTree;
```

cvCreateContourTree

Creates binary tree representation for input contour.

```
void cvCreateContourTree(CvSeq *contour, CvMemStorage* storage, CvContourTree
**tree, double threshold);
```

<i>contour</i>	Pointer to the input contour header.
<i>storage</i>	Pointer to the storage block.
<i>tree</i>	Output pointer to the created tree.
<i>threshold</i>	Value of the threshold.

Discussion

This function creates binary tree representation for the input contour *contour* and returns the pointer to its root. If the parameter *threshold* is less than or equal to 0, the function creates full binary tree representation. If the threshold is more than 0, the function creates representation with the precision *threshold* (the vertices with the interceptive area of its base line less than *threshold*, the tree should not be written to).

cvContourFromContourTree

Restores contour from its binary tree representation.

```
void cvContourFromContourTree (CvContourTree *tree, CvMemStorage* storage,
CvSeq **contour, CvTermCriteria criterion);
```

<i>tree</i>	Pointer to the input tree.
<i>storage</i>	Pointer to the storage block.
<i>contour</i>	Output pointer to the restored contour

criterion Criterion for the definition of the threshold value for contour reconstruction (level of precision).

Discussion

This function restores the contour from its binary tree representation. The parameter *criterion* defines the threshold (level of precision) for the contour restoring. If *criterion.type* = CV_TERMCRIT_ITER, the function restores *criterion.maxIter* tree levels. If *criterion.type* = CV_TERMCRIT_EPS, the function restores the contour as long as *tri_area* > *criterion.epsilon* * *contour_area*, where *contour_area* is the magnitude of the contour area and *tri_area* is the magnitude of the current triangle area. If *criterion.type* = CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, the function restores the contour as long as the one of these conditions is true.

cvMatchContourTrees

Compares two binary tree representations.

```
void cvMatchContourTrees (CvContourTree *tree1, CvContourTree *tree2,
    CvTreeMatchMethod method, double threshold, double* result);
```

<i>tree1</i>	Pointer to the first input tree
<i>tree2</i>	Pointer to the second input tree
<i>method</i>	Method for calculation of the similarity measure, now must be only CV_CONTOUR_TREES_MATCH_I1.
<i>threshold</i>	Value of the compared threshold.
<i>result</i>	Pointer to the matching measure.

Discussion

This function calculates the value of the matching measure for two contour trees. The similarity measure calculates level by level from the binary tree roots. If the total calculating value of the similarity for levels from 0 to the specified is more than the parameter *threshold*, the function stops calculations and value of the total similarity

measure is returned as *result*. If the total calculating value of the similarity for levels from 0 to the specified is less than or equal to *threshold*, the function continues calculation on the next tree level and returns the value of the total similarity measure for the binary trees.

This chapter describes functions from computational geometry field.

Overview

Ellipse Fitting

The fitting of primitive models to the image data is a basic task in pattern recognition and computer vision. A successful solution of this task results in reduction and simplification of the data for the benefit of higher level processing stages. One of the most commonly used models is the ellipse which, being the perspective projection of the circle, is of great importance for many industrial applications.

The representation of general conic by the second order polynomial is

$$F(\vec{a}, \vec{x}) = ax^2 + bxy + cy^2 + dx + ey + f = 0$$

We will denote the vector $\vec{a} = [a, b, c, d, e, f]^T$ and $\vec{x} = [x^2, xy, y^2, x, y, 1]^T$.

$F(\vec{a}, \vec{x})$ is called the “algebraic distance of point (x_0, y_0) to conic $F(a, x)$.

Minimizing the sum of squared algebraic distances $\sum_{i=1} F(\vec{x}_0)^2$ may approach the fitting of conic.

In order to achieve ellipse-specific fitting polynomial coefficients must be constrained. For ellipse they must satisfy $b^2 - 4ac < 0$.

Moreover, we can impose equality constraint $4ac - b^2 = 1$ in order to incorporate coefficients scaling into constraint.

This constraint may be written as a matrix $\vec{a}^T C \vec{a} = 1$.

Finally, the problem could be formulated as minimize $\|D\vec{a}\|^2$ with constraint $\vec{a}^T C \vec{a} = 1$, where D is the $n \times 6$ matrix $[\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n]^T$.

Introducing the Lagrange multiplier, we arrive at the system

$$2D^T D \vec{a} - 2\lambda C \vec{a} = 0, \text{ which could be re-written as}$$

$$\vec{a}^T C \vec{a} = 1$$

$$S \vec{a} = 2\lambda C \vec{a}$$

$$\vec{a}^T C \vec{a} = 1$$

In [1] the system solution is described.

After the system is solved, ellipse center and axis could be extracted.

For more information see [Fitzgibbon95].

Line fitting

M-estimators are used for approximating a set of points with geometrical primitives (e.g., conic section) in cases when the classical least squares method fails. For example, the image of a line from the camera contains noisy data with many outliers, i.e., few points that lie far from the main group, and the least squares method will fail if applied.

The least squares method searches for a parameter set that minimizes the sum of squared distances:

$$m = \sum_i d_i^2,$$

where d_i is the distance (in some meaning) from the i^{th} point to the primitive. If even few points have a large d_i , then the perturbation in the primitive parameter values may be prohibitively big. The solution is to minimize

$$m = \sum_i \rho(d_i),$$

where $\rho(d_i)$ grows slower than d_i^2 . This problem can be reduced to *weighted least squares* [1], which is solved by iterative finding of the minimum of

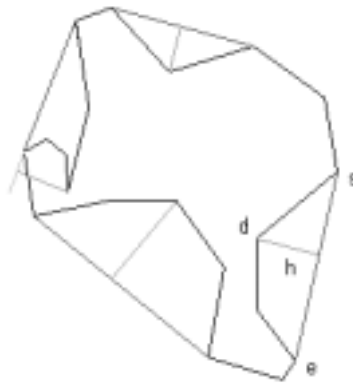
$$m_k = \sum_i W(d_i^{k-1}) d_i^2,$$

where k is the iteration number, d_i^{k-1} is the minimizer of the sum on the previous iteration and $W(x) = \frac{1}{x} \frac{dp}{dx}$. If d_i is a linear function of parameters p_j - $d_i = \sum A_{ij} p_j$ then the minimization vector of the m_k is the eigenvector of $A^{T*}A$ matrix that corresponds to the smallest eigenvalue.

For more information see “[Zhang96] Zhengyou Zhang. Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting, to appear in Image and Vision Computing Journal, 1996.”.

Convexity Defects.

Let (p_1, p_2, \dots, p_n) be a closed simple polygon, or contour, and (h_1, h_2, \dots, h_m) a convex hull. A sequence of contour points exists normally between two consecutive convex hull vertices, this sequence forms the so-called convexity defect for which some useful characteristics can be computed. Computer Vision Library computes only one such characteristic, named “depth” (see the figure below).



The black lines belong to the input contour. The red lines update the contour to its convex hull.

The symbols “*s*” and “*e*” signify the start and the end points of the convexity defect. The symbol “*d*” is a contour point located between “*s*” and “*e*” being the farthestmost from the line that includes the segment “*se*”. The symbol “*h*” stands for the convexity defect depth, i.e., the distance from “*d*” to the “*se*” line.

The structure `CvConvexityDefect` represents the convexity defect.

Example 4-1 `CvConvexityDefect` structure definition

```
typedef struct
{
    CvPoint* start;      //start point of defect
    CvPoint* end;        //end point of defect
    CvPoint* depth_point; //fathermost point
    float    depth;      //depth of defect
} CvConvexityDefect;
```

cvFitEllipse_32f

Fits ellipse to set of 2D points

```
void cvFitEllipse_32f( CvPoint* points, int n, CvBox2D32f* box );
```

<i>points</i>	Pointer to the set of 2D points.
<i>n</i>	Number of points, must be more than or equal to 6.
<i>box</i>	Pointer to the structure for representation of the output ellipse.

Discussion

The function fills the output structure in the following way.

<i>box->center</i>	Point of the center of the ellipse;
<i>box->size</i>	Sizes of two ellipse axes;
<i>box->angle</i>	Angle between (1,0) the vector and the axis of the ellipse of the length <i>box->size.width</i> .

The output ellipse has the property of `box->size.width > box->size.height`.

cvFitLine

Finds lines on binary image.

```
void cvFitLine2D ( CvPoint2D32f* points, int count, CvDisType distType, void*
    param, float reps, float aeps float* line);
```

<i>points</i>	Array of 2D points.
<i>count</i>	Number of points.
<i>distType</i>	Type of the distance used to fit the data to a line (see discussion).
<i>param</i>	Pointer to a user-defined function that calculates the weights for the type <code>IPPI_DIST_USER</code> , or the pointer to a float user-defined metric parameter <i>c</i> for the Fair and Welsch distance types.
<i>reps, aeps</i>	Used for iteration stop criteria (see discussion). If zero, the default value of 0.01 is used.
<i>line</i>	Pointer to the array of 4 floats. When the function exits, the first two elements contain the direction vector of the line normalized to 1, the other two contain coordinates of a point that belongs to the line.

```
void cvFitLine3D ( CvPoint3D32f* points, int count, CvDisType distType, void*
    param, float reps, float aeps float* line);
```

<i>points</i>	Array of 3D points.
<i>count</i>	Number of points.
<i>distType</i>	Type of the distance used to fit the data to a line (see discussion).
<i>param</i>	Pointer to a user-defined function that calculates the weights for the type <code>IPPI_DIST_USER</code> or the pointer to a float user-defined metric parameter <i>c</i> for the Fair and Welsch distance types.
<i>reps, aeps</i>	Used for iteration stop criteria (see discussion). If zero, the default value of 0.01 is used.

line Pointer to the array of 6 floats. When the function exits, the first three elements contain the direction vector of the line normalized to 1, the other three contain coordinates of a point that belongs to the line.

Discussion

Possible distance type values are listed below.

IPPI_DIST_L2	Standard least squares $\rho(x) = x^2$.
IPPI_DIST_L1	
IPPI_DIST_L12	
IPPI_DIST_FAIR	$c = 1.3998$.
IPPI_DIST_WELSCH	$\rho(x) = \frac{c^2}{2} \left[1 - \exp\left(-\left(\frac{x}{c}\right)^2\right) \right]$, $c = 2.9846$.
IPPI_DIST_USER	Uses a user-defined function to calculate the weight The parameter <i>param</i> should point to the function.

The line equation is $[\vec{V} \times (\vec{r} - \vec{r}_0)] = 0$, where $\vec{V} = (line[0], line[1], line[2])$, $\vec{V} = 1$ and $\vec{r}_0 = (line[3], line[4], line[5])$.

In this algorithm \vec{r}_0 is the mean of the input vectors with weights, i.e.,

$$\vec{r}_0 = \frac{\sum_i W(d(\vec{r}_i)) \vec{r}_i}{\sum_i W(d(\vec{r}_i))}.$$

The parameters *reps* and *aeps* are iteration thresholds. If the distance between two values of \vec{r}_0 calculated from two iterations is less than the value of the parameter *reps*, (the distance type IPPI_DIST_C is used in this case) and the angle in radians between two vectors \vec{V} is less than the parameter *aeps*, then the iteration is stopped.

The specification for the user-defined weight function is

```
void user_weight ( float* dist, int count, float* w );
```

dist Pointer to the array of distance values.

count Number of elements.

w Pointer to the output array of weights.

The function should fill the weights array with values of weights calculated from distance values $w[i] = f(d[i])$. $f(x) = \frac{1}{x} \frac{dp}{dx}$ has to be monotone decreasing.

cvProject3D

Project array of 3D points to coordinate axis.

```
void cvProject3D ( CvPoint3D32f*  points3D, int count, CvPoint2D32f* points2D,
                  int xindx, int yindx);
```

<i>points3D</i>	Source array of 3D points.
<i>count</i>	Number of points.
<i>points2D</i>	Target array of 2D points.
<i>xindx</i>	Index of the 3D coordinate from 0 to 2 that will be used as the x-coordinate.
<i>yindx</i>	Index of the 3D coordinate from 0 to 2 that will be used as the y-coordinate

Discussion

This function used with `cvmPerspectiveProject` is intended to provide a general way of projecting a set of 3D points to a 2D plane. The function copies two of the three coordinates specified by the parameters *xindx* and *yindx* of each 3D point to a 2D points array.

cvConvexHull, cvContourConvexHull

Finds convex hull of points set.

```
void cvConvexHull( CvPoint* points, int num_points, CvRect* bound_rect, int
                  orientation, int* hull, int* hullsize );
```

<i>points</i>	Pointer to the set of 2D points.
---------------	----------------------------------

<i>num_points</i>	Number of points.
<i>bound_rect</i>	Pointer to the bounding rectangle of points set (not used).
<i>orientation</i>	Output order of convex hull vertices (CV_CLOCKWISE or CV_COUNTER_CLOCKWISE).
<i>hull</i>	Indices of convex hull vertices in the input array.
<i>hullsize</i>	Number of vertices in convex hull (output parameter).
<pre>void cvContourConvexHull(CvSeq* contour, int orientation, CvMemStorage* storage, CvSeq** hull);</pre>	
<i>contour</i>	Sequence of 2D points.
<i>orientation</i>	Output order of convex hull vertices (CV_CLOCKWISE or CV_COUNTER_CLOCKWISE).
<i>storage</i>	Memory storage where the convex hull must be allocated.
<i>hull</i>	Address of the pointer to the structure <i>CvSeq</i> that will be filled with pointers to convex hull vertices.

Discussion

The function takes an array of points and puts out indices of points that are convex hull vertices. The function uses QUICKSORT for points sorting.

The order in which the vertices appear in the output array is defined by the standard, i.e., bottom-left XY coordinate system

The output structure of the function *cvContourConvexHull* is a sequence of pointers to structures of the *CvPoint* type stored in the input sequence.

cvConvexHullApprox, cvContourConvexHullApprox

Approximately finds convex hull of points set

```
void cvConvexHullApprox( CvPoint* points, int num_points, CvRect* bound_rect,
int bandwidth, int orientation, int* hull, int* hullsize );
```

<i>points</i>	Pointer to the set of 2D points.
---------------	----------------------------------

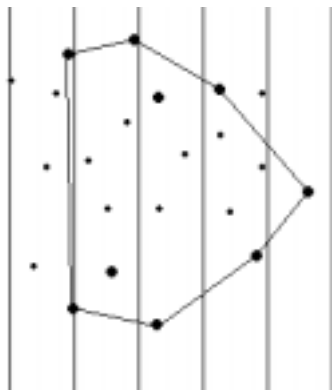
<i>num_points</i>	Number of points.
<i>bound_rect</i>	Pointer to the bounding rectangle of points set (not used).
<i>bandwidth</i>	Width of band used by the algorithm.
<i>orientation</i>	Output order of convex hull vertices (CV_CLOCKWISE or CV_COUNTER_CLOCKWISE).
<i>hull</i>	Indices of convex hull vertices in the input array,
<i>hullsize</i>	Number of vertices in the convex hull (output parameter).

```
void cvContourConvexHullApprox( CvSeq* sequence, int bandwidth, int
orientation, CvMemStorage* storage, CvSeq** hull );
```

<i>contour</i>	Sequence of 2D points.
<i>bandwidth</i>	Bandwidth used by the algorithm.
<i>orientation</i>	Output order of convex hull vertices (CV_CLOCKWISE or CV_COUNTER_CLOCKWISE).
<i>storage</i>	Memory storage where the convex hull must be allocated.
<i>hull</i>	Address of the pointer to the structure <i>CvSeq</i> that will be filled with pointers to convex hull vertices.

Discussion

The following algorithm is used: starting from the extreme left point of the input set, the plane is divided into vertical bands with the specified width (bandwidth). Within every band points with maximal and minimal vertical coordinates are found and all other points are excluded. The next step is finding the exact convex hull of all remaining points (see the figure below).



In the integer points coordinates case, the algorithm can be used to find the exact convex hull; the value of the parameter *bandwidth* must be 1 in this case.

The output structure of the function *cvContourConvexHullApprox* is a sequence of pointers to the structures of the type *CvPoint* stored in the input sequence.

cvCheckContourConvexity

Tests contour convex.

```
int cvCheckContourConvexity( CvSeq* contour );
    contour      Tested contour.
```

Discussion

The function tests if the input is a contour convex or not. The function returns 1 if the contour is convex, 0 otherwise.

cvConvexityDefects

Finds defects of convexity of contour.

```
void cvConvexityDefects( CvSeq* contour, CvSeq* convexhull, CvMemStorage*
    storage, CvSeq** defects );
```

<i>contour</i>	Input contour (sequence of <i>CvPoint</i> structures).
<i>convexhull</i>	Exact convex hull of the input contour. Must be computed by the function <code>cvContourConvexHull</code> .
<i>storage</i>	Memory storage where the sequence of convexity defects must be allocated.
<i>defects</i>	Address of the pointer to the sequence of defects to be created.

Discussion

The function finds all convexity defects of the input contour and creates a sequence of the *CvConvexityDefect* structures.

cvMinAreaRect

Finds circumscribed rectangle of minimal area for given convex contour.

```
void cvMinAreaRect ( CvPoint* points, int n, int left, int bottom, int right,
    int top, CvPoint2D32f* anchor, CvPoint2D32f* vect1, CvPoint2D32f* vect2 );
```

<i>points</i>	Sequence of convex polygon points.
<i>n</i>	Number of input points.
<i>left</i>	Index of the extreme left point.
<i>bottom</i>	Index of the extreme bottom point.
<i>right</i>	Index of the extreme right point.
<i>top</i>	Index of the extreme top point.

<i>anchor</i>	Pointer to one of the output rectangle corners.
<i>vect1</i>	Pointer to the vector that represents one side of the output rectangle.
<i>vect2</i>	Pointer to the vector that represents another side of the output rectangle.

Discussion

The function returns a circumscribed rectangle of the minimal area. The output parameters of this function are the corner of the rectangle and two incident edges of the rectangle (see the figure below).



cvCalcPGH

Calculates pair-wise geometrical histogram for contour.

```
void cvCalcPGH( CvSeq* contour, CvHistogram* hist );
```

contour Input contour.

hist Calculated histogram. The histogram must be two-dimensional.

Discussion

The algorithm considers every pair of the contour edges. The angle between the edges and the minimal/maximal distances are determined for every pair. To determine the distance an edge is treated as the base and the function loops through all other edges; then the next edge becomes base, etc. When the base edge and another edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented. The histogram can be used for contour matching

cvMinEnclosingCircle

Finds minimal enclosing circle for 2D point set.

```
void cvFindMinEnclosingCircle( CvSeq* seq, CvPoint2D32f* center, float* radius
    );
```

<i>seq</i>	Sequence that contains the input point set. Only points with integer coordinates (<i>CvPoint</i>) are supported.
<i>center</i>	Output parameter. Center of the enclosing circle.
<i>radius</i>	Output parameter. Radius of the enclosing circle.

Discussion

The function finds the minimal enclosing circle for the planar point set. “Enclosing” means that all the points from the set are either inside or on the boundary of the circle. Minimal means that there is no enclosing circle with smaller radius.

Fixed filters

Overview

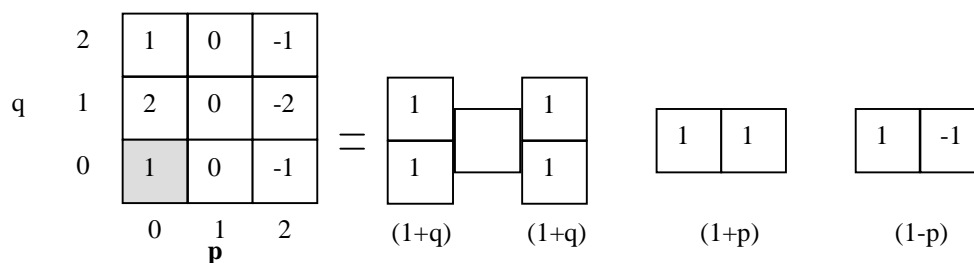
This section describes various fixed filters (primarily derivative operators).

The first, second, and third Sobel Derivative Family.

Consider the first x derivative Sobel operator shown in Figure 1 where the grayed bottom left number indicates the origin in a “ p - q ” coordinate system.

Figure 5-1 First “ x ” derivative Sobel operator

The operator in a “ p - q ” coordinate system could be expressed as a polynomial and decomposed into convolution primitives as shown.



Then, the Sobel operator may be expressed as a polynomial

$1 + 2q + q^2 - p^2 - 2p^2q - p^2q^2 = (1 + q)^2(1 - p^2) = (1 + q)(1 + q)(1 + p)(1 - p)$ and decomposed into convolution primitives as shown in Figure 5-1.

This may be used to express a hierarchy of first x and y derivative Sobel operators as follows:

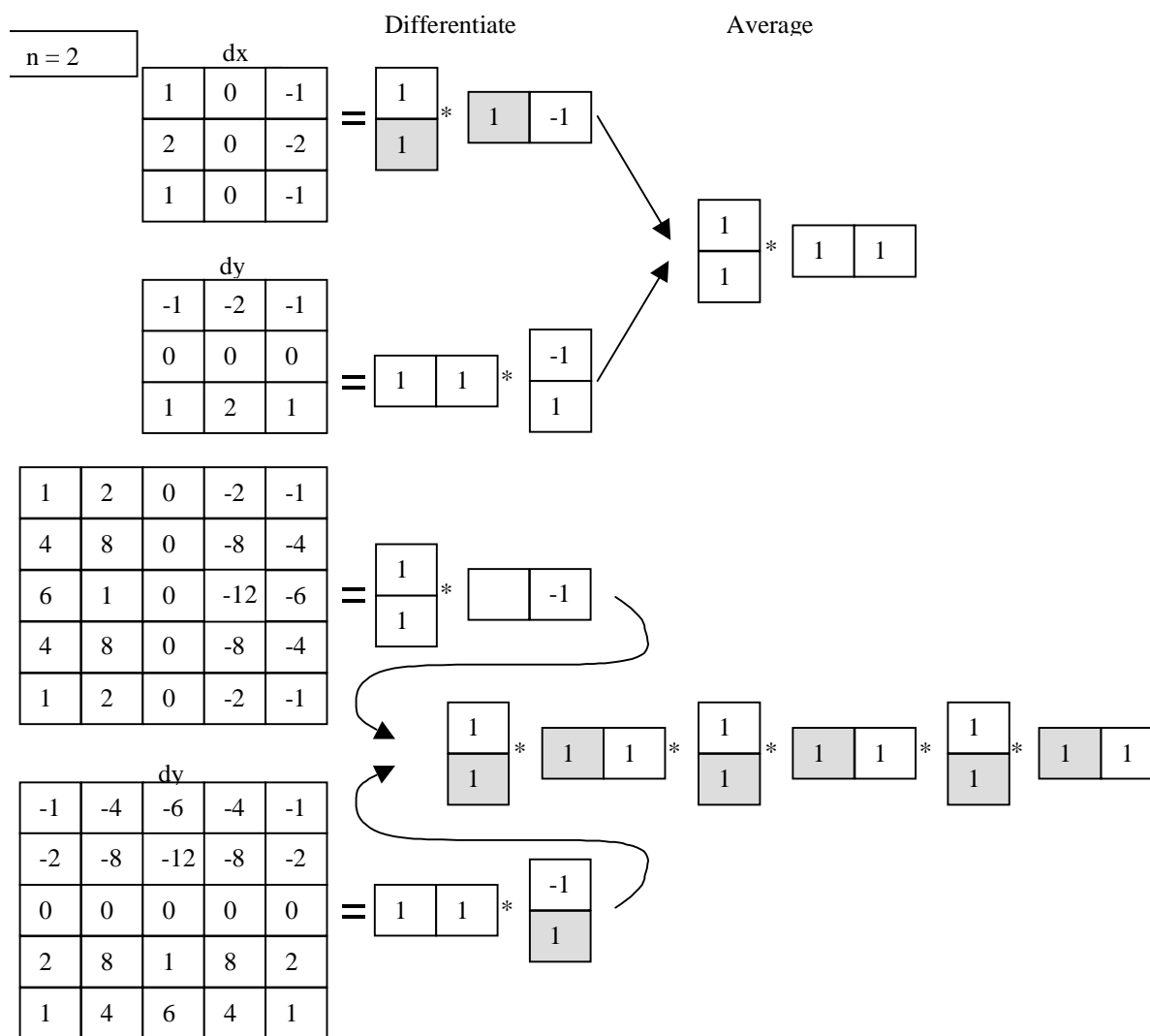
$$\frac{\partial}{\partial x} \Rightarrow (1 + p)^{n-1} (1 + q)^n (1 - p) \quad (1)$$

$$\frac{\partial}{\partial x} \Rightarrow (1 + p)^n (1 + q)^{n-1} (1 - q) \quad (2)$$

for $n > 0$.

Figure 5-2 shows Sobel first derivative filters of equations (1) and (2) for $n = 2, 4$. The Sobel filter may be decomposed into simple “add-subtract” convolution primitives.

Figure 5-2 First derivative Sobel operators for n=2, 4 decomposed into primitive add-subtract convolution operators



Second derivative Sobel operators can be expressed in polynomial decomposition similar to equations (1) and (2). The second derivative equations are:

$$\frac{\partial^2}{\partial x^2} \Rightarrow (1+p)^{n-2}(1+q)^n(1-p)^2 \quad (3)$$

$$\frac{\partial^2}{\partial y^2} \Rightarrow (1+p)^{n-1}(1+q)^{n-2}(1-q)^2 \quad (4)$$

$$\frac{\partial^2}{\partial x \partial y} \Rightarrow (1+p)^{n-1}(1+q)^{n-1}(1-p)(1-q) \quad (5)$$

for $n = 2, 3, \dots$

Figure 5-3 shows the filters that result for $n = 2$ and 4. Just as shown in Figure 5-2, these filters can be decomposed into simple “add-subtract” separable convolution operators as indicated by their polynomial form in the equations.

Figure 5-3 Sobel operator second order derivators for $n = 2$ and 4

The polynomial decomposition is shown above each operator.

$$\delta^2/\delta x^2 = (1+q)^2(1-p)^2$$

1	-2	1
2	-4	2
1	-2	1

$$\delta^2/\delta y^2 = (1+p)^2(1-q)^2$$

1	2	1
-2	-4	-2
1	2	1

$$\delta^2/\delta x \delta y = (1+q)(1+p)$$

-1	0	1
0	0	0
1	0	-1

$$\delta^2/\delta x^2 = (1+p)^2(1+q)^4(1-p)^2$$

1	0	-2	0	1
4	0	-4	0	4
6	0	-12	0	6
4	0	-8	0	4
1	0	-2	0	1

$$\delta^2/\delta y^2 = (1+q)^2(1+p)^4(1-q)^2$$

1	4	6	4	1
0	0	0	0	0
-2	-8	-12	-8	-2
0	0	0	0	0
1	4	6	4	1

$$\delta^2/\delta x \delta y = (1+p)^3(1+q)^3(1-p)(1-q)$$

-1	-2	0	2	1
-2	-4	0	4	2
0	0	0	0	0
2	4	0	-4	-2
1	2	0	-2	-1

Third derivative Sobel operators can also be expressed in the polynomial decomposition form:

$$\frac{\partial^3}{\partial x^3} \Rightarrow (1+p)^{n-3}(1+q)^n(1-p)^3 \quad (6)$$

$$\frac{\partial^3}{\partial y^3} \Rightarrow (1+p)^n(1+q)^{n-3}(1-q)^3 \quad (7)$$

$$\frac{\partial^3}{\partial x^2 \partial y} \Rightarrow (1-p)^2(1+p)^{n-2}(1+q)^{n-1}(1-q) \quad (8)$$

$$\frac{\partial^3}{\partial x \partial y^2} \Rightarrow (1-p)(1+p)^{n-1}(1+q)^{n-2}(1-q)^2 \quad (9)$$

for $n = 3, 4, \dots$. The third derivative filter needs to be applied only for the cases $n = 4$ and general.

Floating point, optimal filter kernels

First derivatives

Table 1 gives coefficients for five increasingly accurate x derivative filters, the y filter derivative coefficients are just column vector versions of the x derivative filters.

Table 5-1 Coefficients for 5 increasingly accurate x derivative filters

Anchor	DX Mask Coefficients					
0	0.74038	-0.12019				
0	0.833812	-0.229945	0.0420264			
0	0.88464	-0.298974	0.0949175	-0.0178608		
0	0.914685	-0.346228	0.138704	-0.0453905	0.0086445	
0	0.934465	-0.378736	0.173894	-0.0727275	0.0239629	-0.00459622

Five increasingly accurate separable x derivative filter coefficients. The table gives half coefficients only. The full table can be obtained by mirroring across the central anchor coefficient. The greater the number of coefficients used, the less distortion from the ideal derivative filter.

Second derivatives

Table 2 gives coefficients for five increasingly accurate x second derivative filters. The y second derivative filter coefficients are just column vector versions of the x second derivative filters.

Table 5-2 Coefficients for five increasingly accurate x second derivative filters

Anchor	DX Mask Coefficients				
-2.20914	1.10457				
-2.71081	1.48229	-0.126882			
-2.92373	1.65895	-0.224751	0.0276655		
-3.03578	1.75838	-0.291985	0.0597665	-0.00827	
-3.10308	1.81996	-0.338852	0.088077	-0.0206659	0.00301915

The table gives half coefficients only. The full table can be obtained by mirroring across the central anchor coefficient. The greater the number of coefficients used, the less distortion from the ideal derivative filter.

Laplacian approximation

The Laplacian may be approximated as the addition of the x and y second derivatives:

$$L = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (10)$$

Thus, any of the equations defined in the sections for second derivatives may be used .

Reference

cvLaplace

Calculates convolution of input image with Laplacian operator.

```
void cvLaplace( IplImage* src, IplImage* dst, Int size);
```

<i>src</i>	Input image.
<i>dst</i>	Destination image.
<i>size</i>	Size of Laplacian kernel.

Discussion

The function calculates the convolution of the input image *src* with the Laplacian kernel of a specified size *size* and stores the result in *dst*.

cvSobel

Calculates convolution of input image with Sobel operator.

```
void cvSobel ( IplImage* src, IplImage* dst, Int dx, Int dy, Int size);
```

<i>src</i>	Input image.
<i>dst</i>	Destination image.
<i>dx</i>	Order of the derivative <i>x</i> .
<i>dy</i>	Order of the derivative <i>y</i> .
<i>size</i>	Size of extended Sobel kernel. -1 is the special value and corresponds to the Scharr filter $1/16[-3, -10, -3; 0, 0, 0; 3, 10, 3]$ (may be transposed).

Discussion

The function calculates the convolution of the input image *src* with a specified Sobel operator kernel and stores the result in *dst*.

Feature detection functions**Overview**

A set of Sobel derivative filters may be used to find edges, ridges, and blobs, especially in a scale-space (image pyramid) situation. Below follows a description of methods in which the filter set could be applied.

- D_x is the first derivative in the x direction just as D_y .
- D_{xx} is the second derivative in the x direction just as D_{yy} .
- D_{xy} is the partial derivative with respect to x and y .
- D_{xxx} is the third derivative in the x direction just as D_{yyy} .
- D_{xxy} and D_{xyy} are the third partials in the x, y directions.

Corner Detection**Method 1**

Corners may be defined as areas where level curves multiplied by the gradient magnitude raised to the power of 3 assume a local maximum

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy} \quad (11)$$

Method 2

Sobel first derivative operators are used to take the x and y derivatives of an image, after which a small region of interest is defined to detect corners in. A 2x2 matrix of the sums of the x and y derivatives is subsequently created as follows:

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix} \quad (12)$$

The eigenvalues are found by solving $\det(C - \lambda I) = 0$, where the lambda is a column vector of the eigenvalues and I is the identity matrix. For the 2x2 matrix of the equation above, the solutions may be written in a closed form:

$$\lambda = \frac{\sum D_x^2 + \sum D_y^2 \pm \sqrt{(\sum D_x^2 + \sum D_y^2)^2 - 4(\sum D_x^2 \sum D_y^2 - (\sum D_x D_y)^2)}}{2} \quad (13)$$

If $\lambda_1, \lambda_2 > \tau$, where τ is some threshold, then a corner is found at that location. This can be very useful for object or shape recognition.

Canny Edge Detector

Stage 1. Image Smoothing

The image data is smoothed by a Gaussian function of width specified by the user parameter.

Stage 2. Differentiation

Assuming two-dimensional convolution at stage 1, the smoothed image data are differentiated with respect to the x and y directions. Next, the gradient of the smooth surface of the convoluted image function could be computed in any direction from the known gradient in any two directions.

From the computed x and y gradient values, the magnitude and the angle of the slope can be calculated from the hypotenuse and arctangent.

In the Computer Vision Library steps 1 and 2 are joined with use of the function `ippiSobel`.

Stage 3. Non-maximum Suppression

Having found the rate of intensity change at each point on the image, the edges must now be placed at the points of maximum, or rather non-maximum must be suppressed. A local maximum occurs at a peak in the gradient function, or alternatively where the derivative of the gradient function is set to zero. However, in this case we need to

suppress the non-maximum perpendicular to the edge direction, rather than parallel to the edge direction, since we expect continuity of the edge strength along an extended contour.

The algorithm starts off by reducing the angle of gradient to one of the four sectors shown on figure 5-4. The algorithm passes the 3x3 neighborhood across the magnitude array. At each point the center element of the neighborhood is compared with its two neighbors along line of the gradient given by the sector value.

If the central value is not greater than neighbors, i.e., non-maximum, it is suppressed.

Figure 5-4 Gradient sectors

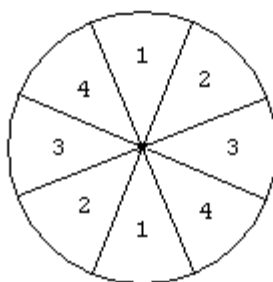


FIGURE MISSING

Stage 4. Edge Thresholding

The thresholder used in the Canny operator uses a method called "hysteresis". Most thresholders use a single threshold limit, which means that if the edge values fluctuate above and below this value, the line will appear broken (commonly referred to as "streaking"). Hysteresis counters streaking by setting an upper and lower edge value

limit. Considering a line segment, if a value lies above the upper threshold limit it is immediately accepted. If the value lies below the low threshold it is immediately rejected. Points which lie between the two limits are accepted if they are connected to pixels which exhibit strong response. The likelihood of streaking is reduced drastically since the line segment points must fluctuate above the upper limit and below the lower limit for streaking to occur. J.Canny recommends the ratio of high to low limit be in the range two or three to one, based on predicted signal-to-noise ratios.

Reference

cvCanny

Implements Canny algorithm for edge detection.

```
void cvCanny( IplImage* img, IplImage* edges, int aperture_size, double
              low_thresh, double high_thresh, );
```

<i>img</i>	Input image.
<i>edges</i>	Image to store the edges found by the function.
<i>aperture_size</i>	Size of the Sobel operator to be used in the algorithm.
<i>low_thresh</i>	Low threshold used for edge searching.
<i>high_thresh</i>	High threshold used for edge searching.

Discussion

The function finds the edges on the input image *img* and stores them into the output image *edges* using the Canny algorithm described above.

cvPreCornerDetect

Calculates two constraint images for corner detection.

```
void cvPreCornerDetect( IplImage* img, IplImage* corners, Int aperture_size );
```

img Input image.

corners Image to store the results.

aperture_size Size of the Sobel operator to be used in the algorithm.

Discussion

The function finds the corners on the input image *img* and stores them into the output image *corners* in accordance with Method 1 for corner detection.

cvCornerEigenValsandVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cvCornerEigenValsAndVecs( IplImage* img, IplImage* eigenvv, Int  
    aperture_size, Int block_size );
```

img Input image.

eigenvv Image to store the results.

aperture_size Derivative operator size (aperture) decreased by 1 in the case of the byte source format. In the case of 32f input this parameter is the number of the fixed float filter used for differencing.

block_size Size of the square block where the corner is searched. The square block where corner is searched will be *blockSize* × *blockSize* pixels.

Discussion

The function takes blocks off each pixel, computes the first derivatives D_x and D_y and the minimal eigenvalue of the matrix.

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix}, \text{ where summation are made over full blocks.}$$

The format of the frame *eigenvv* is the following: for every pixel of the input image the frame contains 6 float values ($\lambda_1, \lambda_2, x_1, y_1, x_2, y_2$).

λ_1, λ_2 are eigen values of the above matrix (not sorted by value).

x_1, y_1 are coordinates of the normalized eigen vector that corresponds to λ_1 .

x_2, y_2 are coordinates of the normalized eigen vector that corresponds to λ_2 .

In a singular matrix or if one of the eigenvalues is much less than another, all six values are set to 0. The Sobel operator is used for differentiation.

cvCornerMinEigenVal

Calculates minimal eigenvalues of image blocks for corner detection.

```
void cvCornerMinEigenVal( IplImage* img, IplImage* eigenv, Int aperture_size,
                          Int block_size );
```

img Input image.

eigenvv Image to store the results.

aperture_size Derivative operator size (aperture) decreased by 1 in the case of the byte source format. In the case of 32f input this parameter is the number of the fixed float filter used for differencing.

block_size Size of the square block where the corner is searched. The square block where corner is searched will be *blockSize* × *blockSize* pixels.

Discussion

The function takes blocks off each pixel, computes the first derivatives D_x and D_y , the eigenvalues and vectors of the matrix.

$$C = \begin{bmatrix} \sum D_x^2 & \sum D_x D_y \\ \sum D_x D_y & \sum D_y^2 \end{bmatrix}, \text{ where summation are made over full blocks.}$$

In a singular matrix eigenvalues are set to 0. The Sobel operator is used for differentiation.

cvFindCornerSubPix

Refines corner locations.

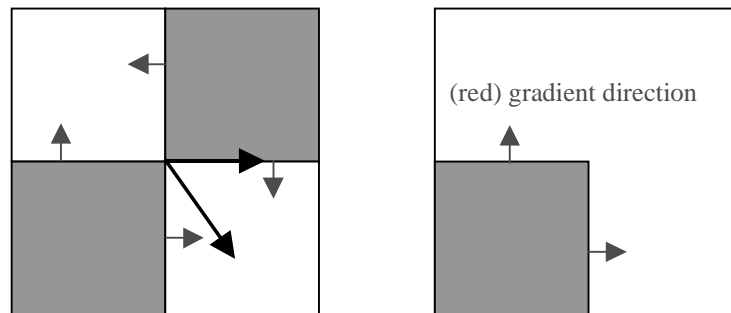
```
void cvFindCornerSubPix( IplImage* img, CvPoint2D32f* corners, int count,
CvSize win, CvSize zeroZone, CvTermCriteria criteria );
```

<i>img</i>	Input raster image.
<i>corners</i>	Initial coordinates of the input corners and refined coordinates on output.
<i>count</i>	Number of corners.
<i>win</i>	Half sizes of the search window. For example, if $win = (5, 5)$, then $5*2 + 1 \times 5*2 + 1 = 11 \times 11$ pixel window to be used.
<i>zeroZone</i>	Half size of the dead region in the middle of the search zone to avoid possible singularities of the autocorrelation matrix. The value of $(-1, -1)$ indicates that there is no such zone.
<i>criteria</i>	Criteria for termination of the iterative process of corner refinement. Iterations may specify a stop when either required precision is achieved or the maximal number of iterations done.

Discussion.

This function iterates to find the accurate sub-pixel location of a corner, or “radial saddle point”, as shown in Figure 5-5 below.

Figure 5-5 Sub-pixel accurate corner



Sub-pixel accurate corner (radial saddle point) locator based on the observation that any vector from q to p is orthogonal to the image gradient.

The core idea of this algorithm is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Thus we have:

Where is the image gradient at one of the points p in a neighborhood of q . We want to find q such that is minimized. We set up a system of equations with the is set to zero:

Where the gradients are summed within a neighborhood (window) of q . Calling the first gradient term G , and the second gradient term b , we get:

Set center the neighborhood window at this new center q and then iterate until it doesn't move more than a set threshold.

cvGoodFeaturesToTrack

Determines strong corners on image.

```
void cvGoodFeaturesToTrack( IplImage* image, IplImage* eigImage, IplImage*  
    tempImage, CvPoint2D32f* corners, int* corner_count, double  
    quality_level, double min_distance );
```

<i>image</i>	Source image (byte, signed byte, or floating-point depth, single channel).
<i>eigImage</i>	Temporary image for minimal eigen vlaues for pixels (floating-point, single channel).
<i>tempImage</i>	Another temporary image (floating-point, single channel).
<i>corners</i>	Output parameter. Found corners.
<i>corner_count</i>	Output parameter. Number of found corners.
<i>quality_level</i>	Specifies minimal accepted quality of image corners (multiplier for the maxmin eigen value).
<i>min_distance</i>	No returned corners can be closer to each other that this limit (Euclidian distance is used).

Discussion

The function finds corners with big eigen values on the image. The function first calculates the minimal eigen value for every pixel of the source image and then performs non-maxima suppression (only local maxima in 3x3 neighborhood remain). The next step is rejecting the corners with the minimal eigen value less than *quality_level**<max_of_min_eigen_vals>. Finally, the function ensures that all found corners are enough distanced from one another by getting two strongest features and checking that the distance between the points is satisfactory. If not, the point is rejected.

Hough Transform

Overview

The Hough Transform (HT) is a popular method of extracting geometric primitives from raster images. The simplest version of the algorithm just detects lines, but it is easily generalized to find more complex features. There are several classes of HT that differ by the image information available. If the image is arbitrary, the Standard Hough Transform (SHT, [1]) should be used.

SHT, like all HT algorithms, considers a discrete set of single primitive parameters. If lines should be detected, then the parameters are ρ and θ , such that the line equation is $\rho = x\cos(\theta) + y\sin(\theta)$.

Here ρ is the distance from the origin to the line, and θ is the angle from the x axis to the perpendicular to the line vector that points from the origin to the line. Every pixel in the image may belong to many lines described by a set of parameters. In other words, we define the “*accumulator*”, which is an integer array $A(\rho, \theta)$, which initially contains only zeroes. For each pixel in the image we increment by 1 all array elements, that correspond to the line that contains the pixel. Then we define a threshold, that enables us to distinguish lines and noise features, and mark all pairs (ρ, θ) for which $A(\rho, \theta)$ is greater than the threshold value. All such pairs characterize detected lines.

Multidimensional Hough Transform (MHT) is a modification of SHT. It performs precalculation of SHT on rough resolution in parameter space and detects the regions of parameters values that possibly have strong support, i.e., correspond to lines in the source image. MHT should be applied to images with few lines and without noise.

[2] presents advanced algorithm for detecting multiple primitives, Progressive Probabilistic Hough Transform (PPHT). The idea is to consider random pixels one by one. Every time the accumulator is changed, it's highest peak is tested on exceeding threshold. If the test succeeds, points that belong to the corridor specified by the peak are removed. If the number of points exceeds the predefined value (minimum line length), then the feature is considered a line, otherwise it is considered a noise. Then the process repeats from the very beginning until no pixel remains in the image. The algorithm improves the result every step, so it can be stopped at any time. [2] claims

that PPHT is easily generalized in almost all cases where SHT could be generalized. The disadvantage of this method is that it does not process correctly some features, for instance, crossed lines, unlike SHT.

For more information see [Matas] and [Trucco98].

Reference

cvHoughLines

Finds lines on binary image, SHT algorithm.

```
void cvHoughLines ( IplImage* src, double rho, double theta int threshold
    float* lines, int linesNumber);
```

<i>src</i>	Source image.
<i>rho</i>	Radius resolution.
<i>theta</i>	Angle resolution.
<i>threshold</i>	Threshold parameter.
<i>lines</i>	Pointer to the array of output lines parameters.
<i>linesNumber</i>	Number of lines. The array should be of size of <i>linesNumber*2 floats</i> .

Discussion

The function `cvHoughLines` should be used if the source image is arbitrary.

cvHoughLinesSDiv

Finds lines on binary image, MHT algorithm.

```
void cvHoughLinesSDiv ( IplImage* src, double rho, int srn, double theta, int
                        stn, int threshold float* lines, int linesNumber);
```

<i>src</i>	Source image.
<i>rho</i>	Rough radius resolution.
<i>srn</i>	Radius accuracy coefficient, ρ/srn is accurate ρ resolution.
<i>theta</i>	Rough angle resolution.
<i>stn</i>	Angle accuracy coefficient, θ/stn is accurate angle resolution.
<i>threshold</i>	Threshold parameter.
<i>lines</i>	Pointer to the array of output lines parameters.
<i>linesNumber</i>	Number of lines, the array should be of size of $linesNumber*2$ floats.

Discussion

The function `cvHoughLinesSDiv` should be applied to images without noise and with few lines.

cvHoughLinesP

Finds lines on binary image, PPHT algorithm.

```
void cvHoughLinesP ( IplImage* src, double rho, double theta, int threshold int
                    lineLength, int lineGap, float* lines, int linesNumber);
```

<i>src</i>	Source image.
<i>rho</i>	Rough radius resolution.
<i>theta</i>	Rough angle resolution.

<i>threshold</i>	Threshold parameter.
<i>linelength</i>	Minimum accepted line length.
<i>lineGap</i>	Maximum length of accepted line gap.
<i>lines</i>	Pointer to the array of output lines parameters.
<i>linesNumber</i>	Number of lines, the array should be of size of <i>linesNumber*2 floats</i> .

Discussion

The function `cvHoughLinesP` works better on noisy images that contain evident lines.

Image Statistics

6

Reference

cvCountNonZero

Counts non-zero pixels on image.

```
int cvCountNonZero (IplImage* image );  
    image           Pointer to the source image.
```

Discussion

The function counts non-zero pixels on the rectangular part of single image plane.

Return values

The function returns a number of non-zero pixels or negative value if bad arguments are passed: bad image header, unsupported image format or `coi` value equal to 0 for three-channel image.

cvSumPixels

Summarizes pixel values on image.

```
double cvSumPixels( IplImage* image );
```

image Pointer to the source image.

Discussion

The function summarizes pixel values on the rectangular part of single image plane.

Return values

The function returns a sum of pixel values in the ROI or a large negative number (`-DBL_MAX`) if bad arguments are passed: bad image header, unsupported image format or `coi` value equal to 0 for three-channel image.

cvMean

Calculates mean pixel value on the image.

```
double cvMean( IplImage* image );
```

image Pointer to the source image.

Discussion

The function calculates the mean pixel value on the rectangular part of single image plane.

Return values

The function returns a mean value or a large negative number (`-DBL_MAX`) if bad arguments are passed: bad image header, unsupported image format or `coi` value equal to 0 for three-channel image.

cvMeanMask

Calculates mean pixel value on image selected area.

```
double cvMeanMask( IplImage* image, IplImage* mask );
```

<i>image</i>	Pointer to the source image.
<i>mask</i>	Pointer to the single-channel mask image.

Discussion

The function calculates mean pixel value on the arbitrary part of single image plane. The mask image must have one-channel IPL_DEPTH_8U type.

Return values

The function returns a mean value or a large negative number (`-DBL_MAX`) if bad arguments are passed: bad source or mask image.

cvMean_StdDev

Calculates mean and standard deviation of pixel values on image.

```
void cvMean_StdDev( IplImage* image, double* mean, double* stddev );
```

<i>image</i>	Pointer to the source image.
<i>mean</i>	Pointer to returned mean.
<i>stddev</i>	Pointer to returned standard deviation.

Discussion

The function calculates mean and standard deviation of pixel values on the rectangular part of single image plane.

cvMean_StdDevMask

Calculates mean and standard deviation of pixel values on selected image area.

```
void cvMean_StdDevMask( IplImage* image, IplImage* mask, double* mean, double*
    stddev );
```

<i>image</i>	Pointer to the source image.
<i>mask</i>	Pointer to the single-channel mask image.
<i>mean</i>	Pointer to returned mean.
<i>stddev</i>	Pointer to returned standard deviation.

Discussion

The function calculates mean and standard deviation of pixel values on the arbitrary part of single image plane.

cvMinMaxLoc

Finds minimum and maximum pixel values on image and their positions.

```
void cvMinMaxLoc( IplImage* image, double* minVal, double* maxVal, CvPoint*
    minLoc, CvPoint* maxLoc );
```

<i>image</i>	Pointer to the source image.
<i>minVal</i>	Pointer to returned minimum value.
<i>maxVal</i>	Pointer to returned maximum value.
<i>minLoc</i>	Pointer to returned minimum location.
<i>maxLoc</i>	Pointer to returned maximum location.

Discussion

The function finds minimum and maximum pixel values on the rectangular part of single image plane their positions. If there are several minimums/maximums on the image, the function returns the most top-left locations.

cvMinMaxLocMask

Finds minimum and maximum pixel values on selected image area and their positions.

```
void cvMinMaxLocMask( IplImage* image, IplImage* mask, double* minVal, double* maxVal, CvPoint* minLoc, CvPoint* maxLoc );
```

<i>image</i>	Pointer to the source image.
<i>mask</i>	Pointer to the single-channel mask image.
<i>minVal</i>	Pointer to returned minimum value.
<i>maxVal</i>	Pointer to returned maximum value.
<i>minLoc</i>	Pointer to returned minimum location.
<i>maxLoc</i>	Pointer to returned maximum location.

Discussion

The function finds minimum and maximum pixel values on the arbitrary part of a single image plane and their positions. If there are several minimums/maximums on the image, the function returns the most top-left locations. If the specified area is empty (mask is filled with zeros) then $minLoc = maxLoc = \{0, 0\}$, $minVal = maxVal = 0$.

cvNorm

Calculates norms, difference norms and relative difference norms of one or two images

```
double cvNorm( IplImage* imgA, IplImage* imgB, int normType );
```

imgA Pointer to the first source image
imgB Pointer to the second source image if any, NULL otherwise.
normType Type of norm (see Discussion).

Discussion

The function calculates images norms defined below. If *imgB* = NULL, the following three norm types of image A are calculated:

NormType = CV_C: $\|A\|_C = \max(|A_{ij}|)$,

NormType = CV_L1: $\|A\|_{L_1} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij}|$,

NormType = CV_L2: $\|A\|_{L_2} = \sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} A_{ij}^2}$.

If *imgB* ≠ NULL, the difference or relative difference norms are calculated:

NormType = CV_C: $\|A-B\|_C = \max(|A_i - B_i|)$,

NormType = CV_L1: $\|A-B\|_{L_1} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij} - B_{ij}|$,

NormType = CV_L2: $\|A-B\|_{L_2} = \sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (A_{ij} - B_{ij})^2}$,

$$\begin{aligned}
\text{NormType} = \text{CV_RELATIVEC}: \quad \|A - B\|_C / \|B\|_C &= \frac{\max(|A_{ij} - B_{ij}|)}{\max(|B_{ij}|)}, \\
\text{NormType} = \text{CV_RELATIVEL1}: \quad \|A - B\|_{L_1} / \|B\|_{L_1} &= \frac{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |A_{ij} - B_{ij}|}{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} |B_{ij}|}, \\
\text{NormType} = \text{CV_RELATIVEL2}: \quad \|A - B\|_{L_2} / \|B\|_{L_2} &= \frac{\sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (A_{ij} - B_{ij})^2}}{\sqrt{\sum_{i=1}^{N_x} \sum_{j=1}^{N_y} (B_{ij})^2}}.
\end{aligned}$$

Return values

The function returns the value of the norm or a large negative number (in case there is an error).

cvNormMask

Calculates norms, difference norms and relative difference norms of one or two images with mask account.

```
double cvNormMask( IplImage* imgA, IplImage* imgB, IplImage* mask, int
    normType );
```

<i>imgA</i>	Pointer to the first image
<i>imgB</i>	Pointer to the second image if any, <code>NULL</code> otherwise.
<i>mask</i>	Pointer to the mask image
<i>normType</i>	Type of norm (see Discussion).

Discussion

The definitions of the different norm types have been given above (see Discussion in the previous subdivision). Image elements A_{ij} , B_{ij} are taken into account only if $mask_{ij} \neq 0$.

Return values

Returns the value of the norm or a large negative number in case there is an error.

cvMoments

Calculates all moments up to third order of image plane and fills moment state structure.

```
void cvMoments( IplImage* image, CvMoments* moments, int is_binary );
```

<i>image</i>	Pointer to the image (or to top-left corner of its ROI).
<i>moments</i>	Pointer to returned moment state structure.
<i>is_binary</i>	If the flag is non zero, all the zero pixel values are treated as zeroes, all other treated as ones.

Discussion

The function calculates moments up to the third order and writes the result to the moment state structure. This structure is used then to retrieve certain spatial, central, or normalized moment or to calculate H_u moments.

cvGetSpatialMoment, cvGetCentralMoment, cvGetNormalizedCentralMoment

Retrieves spatial, central or normalized central moment from the moment state structure.

```
double cvGetSpatialMoment( CvMoments* moments, int x_order, int y_order );
double cvGetCentralMoment( CvMoments* moments, int x_order, int y_order );
double cvGetNormalizedCentralMoment( CvMoments* moments, int x_order, int
    y_order );
```

<i>moments</i>	Pointer to the moment state structure.
<i>x_order</i>	Order <i>x</i> of required moment.
<i>y_order</i>	Order <i>y</i> of required moment
	(0 ≤ <i>x_order</i> , <i>y_order</i> ; <i>x_order</i> + <i>y_order</i> ≤ 3).

Discussion

The function `cvGetSpatialMoment` retrieves the spatial moment, which is defined as:

$$M_{x_order, y_order} = \sum_{x, y} I(x, y) x^{x_order} y^{y_order}, \text{ where}$$

$I(x, y)$ is the intensity of the pixel (x, y) .

The function `cvGetCentralMoment` retrieves the central moment, which is defined as:

$$\mu_{x_order, y_order} = \sum_{x, y} I(x, y) (x - \bar{x})^{x_order} (y - \bar{y})^{y_order}, \text{ where}$$

$I(x, y)$ is the intensity of pixel (x, y) , \bar{x} is the *x* coordinate of the mass center, \bar{y} is the *y* coordinate of the mass center:

$$\bar{x} = \frac{M_{1,0}}{M_{0,0}}, \bar{y} = \frac{M_{0,1}}{M_{0,0}}$$

The function `cvGetNormalizedCentralMoment` retrieves the normalized central moment, which is defined as:

$$\eta_{x_order, y_order} = \frac{\mu_{x_order, y_order}}{M_{0,0}^{((x_order + y_order)/2 + 1)}}$$

Return values

The function returns the required moment, or a large negative number (`-DBL_MAX`) if a null pointer is passed, or either `x_order`, or `y_order` is invalid.

cvGetHuMoments

Calculates seven moment invariants from the moment state structure.

```
void cvGetHuMoments( CvMoments* moments, CvHuMoments* HuMoments );
```

moments Pointer to the moment state structure.

HuMoments Pointer to the moment state structure.

Discussion

The function calculates seven `Hu` invariants using the following formulae:

$$h_1 = \eta_{20} + \eta_{02},$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2,$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2,$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2,$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \quad , \\ + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$h_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}),$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

These values are proved to be invariants to the image scale, rotation, and reflection except the first one, which sign is changed by reflection.

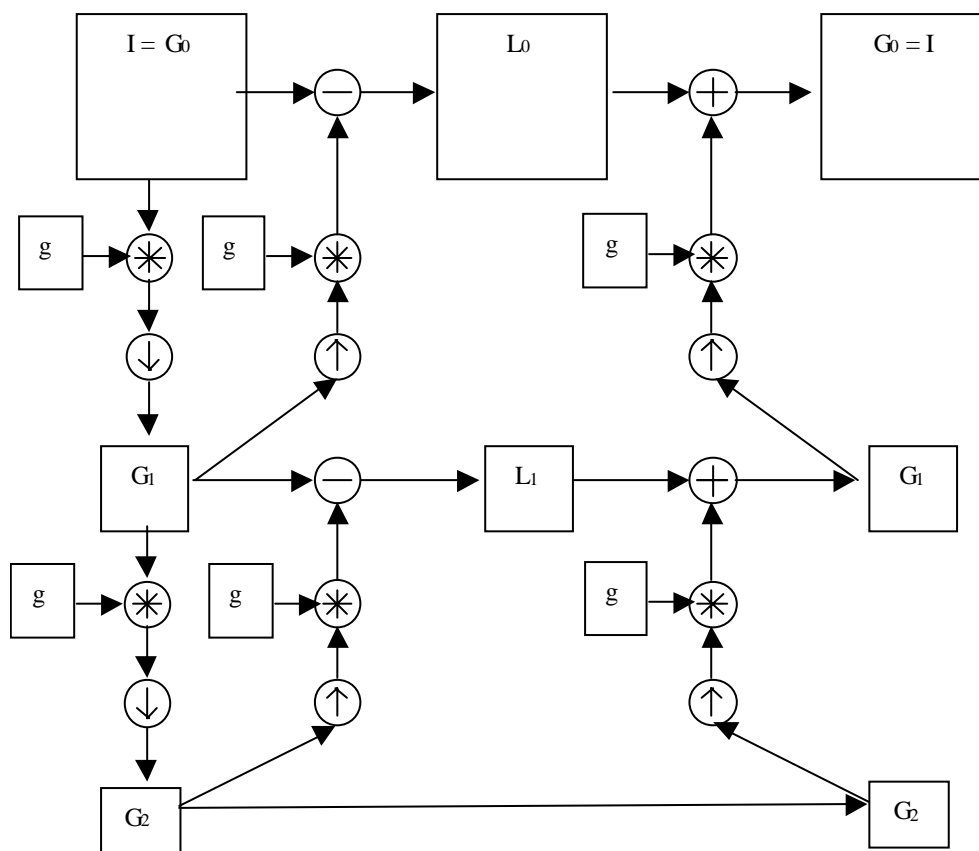
These functions support generation and reconstruction of Gaussian and Laplacian Pyramids.

Overview

Figure 7-1 shows the basics of creating Gaussian or Laplacian pyramids. The original image G_0 is convolved with a Gaussian, then downsampled to get the reduced image G_1 . This process can be continued as far as desired or until the image size is one pixel.

The Laplacian pyramid can be built from a Gaussian pyramid as follows: Laplacian level “ k ” can be built by up sampling the lower level image G_{k+1} . The “missing” pixels resulting from up sampling are interpolated by convolving the image with a Gaussian kernel “ g ” and the resulting image is subtracted from the Gaussian smoothed image G_k . To rebuild the original image, the process is reversed as shown at right Figure 7-1.

Figure 7-1 A three-level Gaussian and Laplacian pyramid.



The Gaussian image pyramid on the left is used to create the Laplacian pyramid in the center, which is used to reconstruct the Gaussian pyramid and the original image on the right. I is the original image, G is the Gaussian image, L is the Laplacian image. Subscripts denote level of the pyramid. g is a Gaussian kernel used to convolve the image before down sampling or after up sampling.

Image segmentation by pyramid

Pyramid based image processing techniques are now widely used in computer vision. The pyramid provides a hierarchical smoothing, segmentation, and hierarchical computing structure that supports fast analysis and search algorithms.

Burt suggested a pyramid-linking algorithm as an effective implementation of a combined segmentation and feature computation algorithm. This algorithm finds connected components without preliminary threshold (i.e., it works on grayscale image). It is an iterative algorithm.

Burt's algorithm includes the following steps:

1. Computation of the Gaussian pyramid;
2. Segmentation by pyramid-linking;
3. Averaging of linked pixels.

Steps 2 and 3 are repeated iteratively until a stable segmentation result is reached.

After computation of the Gaussian pyramid a son-father relationship is defined between nodes (pixels) in adjacent levels. Let's define the next attributes for every node (i, j) on the level l of the pyramid:

$c[i, j, l][t]$ is the value of the local image property (intensity, for example) ;

$a[i, j, l][t]$ is the area over which the property has been computed;

$p[[i, j, l][t]$ is pointer to the node's father, which is in the level $l+1$;

$s[i, j, l][t]$ is the segment property, the average value for the entire segment containing the node.

t is the iteration number ($t \geq 0$). For $t = 0$, $c[i, j, l][0] = G_{i, j}^l$.

For every node (i, j) at level l there are 16 candidate son nodes at level $l-1$ (i', j') , where

$$i' \in \{2i-1, 2i, 2i+1, 2i+2\}, j' \in \{2j-1, 2j, 2j+1, 2j+2\} \quad (8.1)$$

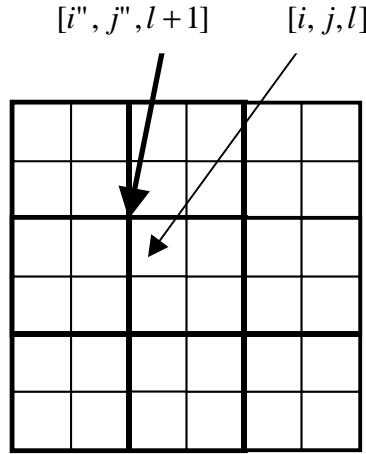
For every node (i, j) at level l there are 4 candidate father nodes at level $l+1$ (i'', j'') , (Figure 7-2), where

$$i'' \in \{(i-1)/2, (i+1)/2\}, j'' \in \{(j-1)/2, (j+1)/2\} \quad (8.2)$$

Son-father links are established for all nodes below the top of pyramid for every iteration t . Let $d[n][t]$ be the absolute difference between the c value of the node (i, j) at the level l and its n^{th} candidate father, then

$$p[i, j, l][t] = \underset{1 \leq n \leq 4}{\operatorname{argmin}} d[n][t] \quad (8.3)$$

Figure 7-2 Connections between adjacent pyramid levels



After the son-father relationship is defined for the t , c , and a value are computed from bottom to the top for the $0 \leq l \leq n$ as

$$a[i, j, 0][t] = 1, \quad c[i, j, 0][t] = c[i, j, 0][0], \quad a[i, j, l][t] = \sum a[i', j', l-1][t].$$

where sum is over this sons node (i, j) as indicated by the links p in (8.3).

If $a[i, j, l][t] > 0$ then $c[i, j, l][t] = \sum ([i', j', l-1][t] \cdot c[i', j', l-1][t]) / a[i, j, l][t]$, but if $a[i, j, 0][t] = 0$ so the node has no sons, $c[i, j, 0][t]$ is set to the value of one of its candidate sons selected at random. No segment values are calculated in the top down order. The value of the initial level L is an input parameter of the algorithm. At level L the segment value of each node is set equal to its local property value:

$$s[i, j, L][t] = c[i, j, L][t]$$

For lower levels $l < L$ each node value is just that of its father $s[i, j, l][t] = c[i'', j'', l + 1][t]$

Here node (i'', j'') is the father of (i, j) , as established in (8.3).

After this the current iteration t finishes and the next iteration $t + 1$ begins. Any changes in pointers in the next iteration will result in changes in the values of local image properties.

The iterative process is continued until no changes occur between two successive iterations.

The choice of L only determines the maximum possible number of segments. If the number of segments less than the numbers of nodes at level L , the values of $c[i, j, L][t]$ are clustered into a number of groups equal to the desired number of segments. The c value for each node in the group is replaced by the group average value, computed from the c values of its members, weighted by their areas a .

Data Structures

The pyramid functions use the data structure `IplImage` for image representation and the data structure `CvSeq` for the sequence of the connected components representation. Every element of this sequence is the data structure `CvConnectedComp` for the single connected component representation in memory.

The C language definition for the `CvConnectedComp` structure is given below.

Example 7-1 `CvConnectedComp` structure Definition

```
typedef struct CvConnectedComp
{
    double area;           /* area of the segmented
                           component */
    float value;           /* gray scale value of the
                           segmented component */
    CvRect rect;          /* ROI of the segmented component
                           */
} CvConnectedComp;
```

Reference

cvPyrDown

Down-samples the image.

```
void cvPyrDown(IplImage* src, IplImage* dst, IplFilter filter);
```

<i>src</i>	Pointer to the source image.
<i>dst</i>	Pointer to the destination image.
<i>filter</i>	Type of the filter which used for convolution (only IPL_GAUSSIAN_5x5 is currently supported).

Discussion

The function performs down-sampling step of Gaussian pyramid decomposition. First it convolves source image with the specified filter and then down-samples the image by rejecting even rows and columns. So the destination image will be four times smaller than the source image.

cvPyrUp

Up-samples the image.

```
void cvPyrUp( IplImage* pSrc, IplImage* pDst, IplFilter filter);
```

<i>src</i>	Pointer to the source image.
<i>dst</i>	Pointer to the destination image.
<i>filter</i>	Type of the filter which used for convolution (only IPL_GAUSSIAN_5x5 is currently supported).

Discussion

The function performs up-sampling step of Gaussian pyramid decomposition. First it up-samples the source image by injecting even zero rows and columns and then convolves result with the specified filter multiplied by 4 for interpolation. So the destination image will be four times larger than the source image.

cvPyrSegmentation

Implements image segmentation by pyramids.

```
void cvPyrSegmentation(IplImage* src_image, IplImage* dst_image, CvMemStorage*
    storage, CvSeq** comp, int level, double threshold1, double threshold2);
```

<i>src_image</i>	Pointer to the input image data,
<i>dst_image</i>	Pointer to the output segmented data,
<i>storage</i>	Storage, where the sequence of connected components will be stored.
<i>comp</i>	Pointer to the output sequence of the segmented components,
<i>level</i>	Maximum level of the pyramid for the segmentation,
<i>threshold1</i>	Error threshold for the links establishing,
<i>threshold2</i>	Error threshold for the segments clustering.

Discussion

This function implements image segmentation by pyramids. The pyramid builds up to level *level*. The links between any pixel *a* on the level *i* and its candidate father pixel *b* on the adjacent level establishes if $\rho(c(a), c(b)) < threshold1$. After the segments components built segments clustering implements. Any two segments *A* and *B* belong the same cluster if $\rho(c(A), c(B)) < threshold2$. The input image has only one channel then $\rho(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then $\rho(c^1, c^2) = 0,3 \cdot (c_r^1 - c_r^2) + 0,59 \cdot (c_g^1 - c_g^2) + 0,11 \cdot (c_b^1 - c_b^2)$. It is necessary to note that one cluster may associates more than one connected components.

Input *src_image* and output *dst_image* have to have the identical `IPL_DEPTH_8U` depth and identical number of channels (1 or 3).

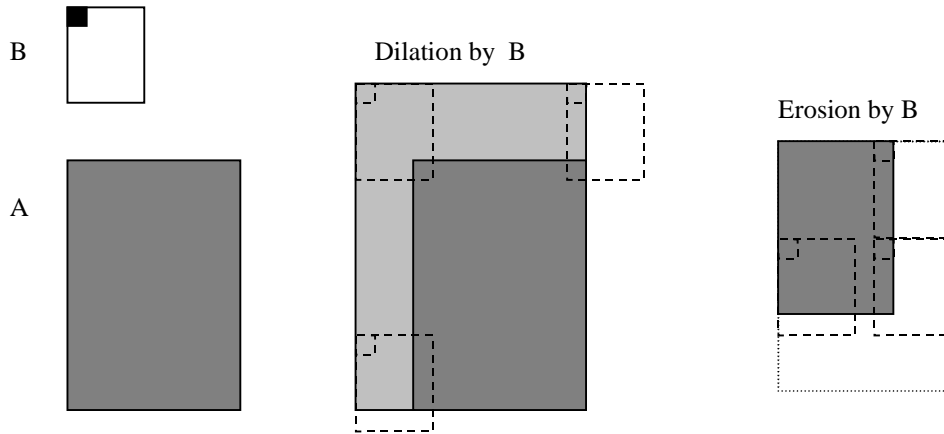
Morphology

8

This chapter describes an expanded set of morphological operators that support segmentation, boundary finding, region filling, and connected components finding.

Overview

Mathematical Morphology is a set-theoretic method of image analysis first developed by Matheron and Serra at the Ecole des Mines, Paris.. The two basic morphological operations are erosion (thinning) and dilation (thickening). All operations involve an image A (object of interest) and a kernel element B called the *structuring element*. The image and structuring element could be in any number of dimensions, but the most common use is with a 2D binary image, or with a 3D gray scale image. B is most often a square or a circle, but could be any shape. Just like in convolution, B is a kernel or template with an anchor point. In Figure 8-1, B is rectangular with an anchor point at upper left. Figure 8-1 shows Dilation and Erosion of object A by B .

Figure 8-1 Dilation and Erosion by B.

If we call the 2D transpose or reflection of B , \bar{B} , and B_t the translation of B around the image, then dilation of object A by structuring element B is

$$A \oplus B = \{t: \bar{B}_t \cap A \neq \emptyset\}$$

What this says is every pixel is in the set if the intersection is not null. That is, a pixel under the anchor point of B is marked on if at least one pixel of B is inside of A .

$$A \oplus nB$$

Indicates the dilation is done n times.

Erosion of object A by structuring element B is:

$$A \ominus B = [t: B_t \subseteq A]$$

That is, a pixel under the anchor of B is marked on if B is entirely within A .

$$A \ominus nB$$

Indicates the erosion is done n times. A useful application of this is that the boundary of A , ∂A can be found by:

$$\partial A = A - (A \ominus nB)$$

Opening of A by B is:

$$A \bullet B = (A \ominus nB) \oplus nB$$

Closing of A by B is:

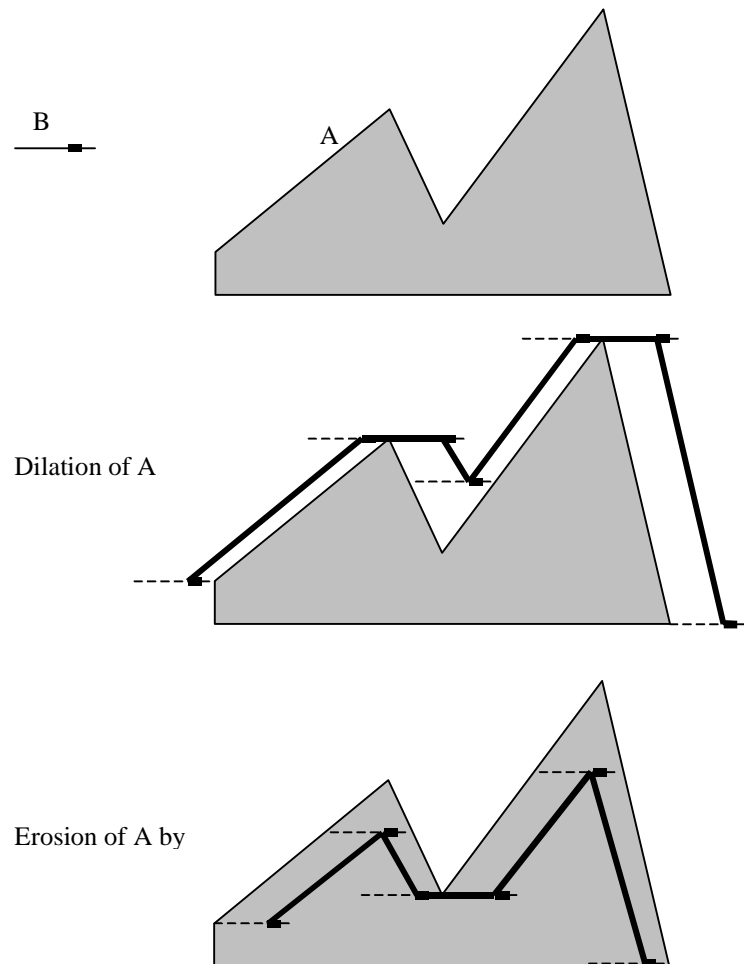
$$A \bullet B = (A \oplus nB) \ominus nB$$

where $n > 0$.

Flat structuring elements for gray scale

Erosion and Dilation can be done in “3D”, i.e., with gray levels. 3D structuring elements can be used, but the simplest and the best way is to use a flat structuring element B as shown in Figure 8-2. In the figure, B has an anchor slightly to the right of center as shown by the dark mark on B . Figure 8-2 is a 1D cross-section of a gray level image A . Dilation and Erosion are shown.

Figure 8-2 1D cross section of Dilation and Erosion of gray scale A by flat structuring element B.



In Figure 8-1, dilation is mathematically

$\sup_{y \in B_c} A$,
and erosion is

$$\inf_{y \in B_c} A \quad .$$

Open and close gray level with a flat structuring element

The typical position of the anchor of the structuring element B for opening and closing is in the center. Subsequent opening and closing could be done in the same manner as in the equations above to smooth off jagged objects (opening tends to cut off peaks and closing tends to fill in valleys).

Morphological Gradient function

A morphological gradient may be taken with the flat gray scale structuring elements as follows:

$$\text{grad}(A) = \frac{(A \oplus B_{\text{flat}}) - (A \ominus B_{\text{flat}})}{2}$$

Top hat and black hat

Top hat (TH) is a function that isolates bumps and ridges from gray scale objects, i.e., it can detect areas that are lighter than the surrounding neighborhood of A and smaller compared to the structuring element. The function subtracts the gray scale object A from the opened version of A :

$$TH_B(A) = A - (A \bullet nB_{\text{flat}})$$

Black Hat (TH^d) is the dual function of Top Hat in that it isolates valleys and “cracks” off of a gray scale object A , i.e., the function detects dark and thin areas by subtracting the closed image A from A :

$$TH_B^d(A) = (A \bullet nB_{\text{flat}}) - A$$

Both Top Hat and Black Hat operations are often followed by thresholding.

Reference

cvCreateStructuringElementEx

Creates structuring element.

```
IplConvKernel* cvCreateStructuringElementEx(int nCols, int nRows, int anchorX,
int anchorY, CvElementShape shape, int* values );
```

<i>nCols</i>	Number of columns in the structuring element.
<i>nRows</i>	Number of rows in the structuring element.
<i>anchorX</i>	Relative horizontal offset of the anchor point.
<i>anchorY</i>	Relative vertical offset of the anchor point.
<i>shape</i>	Specifies the concrete shape of the structuring element. May have the following values: <ul style="list-style-type: none"> • CV_SHAPE_RECT, a rectangular element • CV_SHAPE_CROSS, a cross-shaped element • CV_SHAPE_ELLIPSE, an elliptic element • CV_SHAPE_CUSTOM, a user-defined element. In this case the parameter <i>values</i> specifies the mask, i.e., which neighbors of the pixel must be considered.
<i>values</i>	Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non zero values indicate points that belong to the element. If the pointer is NULL, then all values are considered non zero (rectangular element case). This parameter is considered only if the shape is IPPI_SHAPE_CUSTOM.

Discussion

The function allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the following morphological operations.

Return values

This function returns the structuring element.

cvReleaseStructuringElement

Deletes structuring element.

```
void cvReleaseStructuringElement(IplConvKernel** ppElement);
```

ppElement Pointer to the deleted structuring element.

Discussion

This function releases the structure `IplConvKernel` that is needless already. If **ppElement* is `NULL`, the function does nothing.

cvErode

Erodes image by using arbitrary structuring element.

```
void cvErode( IplImage* src, IplImage* dst, IplConvKernel* B, int iterations);
```

src Source image.

dst Destination image.

B Structuring element used for erosion. If `NULL`, a 3x3 rectangular structuring element is used.

iterations Number of times erosion is applied.

Discussion

The function erodes the source image. The function takes the pointer to the structuring element, consisting of zeros and minus ones and the minus ones determine neighbors of each pixel from which the maximum is taken and put to the corresponding destination pixel. The function supports the in-place mode when the source and destination pointers are the same. Dilation can be applied several times (*iterations* parameter). Erosion on a color image means independent transformation of all channels.

These functions support ROI.

cvDilate

Dilates image by using arbitrary structuring element.

```
void cvDilate( IplImage* pSrc, IplImage* pDst, IplConvKernel* B, int
               iterations);
```

<i>pSrc</i>	Source image.
<i>pDst</i>	Destination image.
<i>B</i>	Structuring element used for dilation. If <code>NULL</code> , a 3x3 rectangular structuring element is used.
<i>iterations</i>	Number of times erosion is applied.

Discussion

The function dilates the source image. The function takes the pointer to the structuring element, consisting of zeros and minus ones; the minus ones determine neighbors of each pixel from which the minimum is taken and put to the corresponding destination pixel. The function supports the in-place mode when the source and destination pointers are the same. Erosion can be applied several times (*iterations* parameter). Erosion on a color image means independent transformation of all channels.

The function performs dilation of the source image. It takes pointer to the structuring element that consists of zeros and minus ones and the minus ones determine neighbors of each pixel from which the maximum is taken and put to the corresponding destination pixel. Function supports in-place mode. Dilation can be applied several times (*iterations* parameter). Dilation on color image means independent transformation of all channels.

These functions support ROI.

cvMorphologyEx

Performs advanced morphological transformations.

```
void cvMorphologyEx( IplImage* src, IplImage* dst, IplImage* temp,
IplConvKernel* B, CvMorphOp op, int iterations );
```

<i>src</i>	Source image.
<i>dst</i>	Destination image.
<i>temp</i>	Temporary image, required in some cases.
<i>B</i>	Structuring element.
<i>op</i>	Type of morphological operation: <ul style="list-style-type: none"> • CV_MOP_OPEN, opening; • CV_MOP_CLOSE, closing; • CV_MOP_GRADIENT, morphological gradient; • CV_MOP_TOPHAT, top hat; • CV_MOP_BLACKHAT, black hat/ (See discussion above for description of these operations).
<i>iterations</i>	Number of times erosion and dilation are applied during the complex operation.

Discussion

The function performs advanced morphological transformations. The function uses `cvErode` and `cvDilate` to perform more complex operations. `temp` must be non `NULL` and point to the image with the same size and same format as `src` (and `dst`) when `op` is `CV_MOP_GRADIENT`, or when `op` is `CV_MOP_TOPHAT` or `op` is `CV_MOP_BLACKHAT` and `src` is equal to `dst` (in-place operation).

Background Differencing

9

This chapter describes Background Differencing functions group.

Reference

cvAcc

Calculates sum of two images.

```
void cvlAcc( IplImage* img, IplImage* sum);
```

img Input image.

sum Accumulating image.

Discussion

The function adds a new image (*img*) to the cumulative sum (*sum*).

cvAccMask

Calculating sum of two images using mask.

```
void cvAccMask( IplImage* img, IplImage* sum, IplImage* mask);
```

img Input image.

sum Accumulating image.

mask Mask image.

Discussion

The function adds the pixel values of the input image *img* to the pixel values of the cumulative image *sum* if the corresponding pixel values in the mask are nonzero, or does nothing otherwise.

cvSquareAcc

Evaluates square of source image and adds it to destination image.

```
void cvSquareAcc( IplImage* img, IplImage* sq_sum);
```

img Input image.
sq_sum Accumulating image.

Discussion

The function adds the square of the new image (*img*) to the cumulative sum (*sq_sum*) of image squares.

cvSquareAccMask

Calculates sum of two image squares using mask.

```
void cvSquareMask( IplImage* img, IplImage* sq_sum, IplImage* mask);
```

img Input image.
sq_sum Accumulating image.
mask Mask image.

Discussion

The function adds the squares of pixel values of the input image *img* to the values of the cumulative image *sum* if the corresponding pixel values in the mask are nonzero, or does nothing otherwise.

cvMultiplyAcc

Evaluates product of two input images and adds it to destination image.

```
void cvMultiplyAcc( IplImage* imgA, IplImage* imgB, IplImage* acc);
```

<i>imgA</i>	First input image.
<i>imgB</i>	Second input image.
<i>acc</i>	Accumulating image.

Discussion

The function multiplies input *imgA* by *imgB* and adds the result to the cumulative sum (*acc*) of image products.

cvMultiplyAccMask

Evaluates product of two input images and adds it to destination image using mask.

```
void cvMultiplyAccMask( IplImage* imgA, IplImage* imgB, IplImage* acc,  
                        IplImage* mask);
```

<i>imgA</i>	First input image.
<i>imgB</i>	Second input image.
<i>acc</i>	Accumulating image.

mask Mask image.

Discussion

The function adds the products of pixel values of input images *imgA* and *imgB* to the values of cumulative image sum, if the corresponding pixel values in the mask are nonzero, or does nothing otherwise.

cvRunningAvg

Calculates weighted sum of two images.

```
void cvRunningAvg ( IplImage* imgY, IplImage* imgU, double alpha);
```

imgY Input image.
imgU Destination image.
alpha Weight of input image.

Discussion

Once a statistical model is available, there is often a need to update the value slowly to account for slowly changing lighting etc. This can be done by using a simple adaptive filter:

$$\mu_t = \alpha y + (1 - \alpha)\mu_{t-1} \quad (14)$$

where μ (*imgU*) is the updated value, $0 \leq \alpha \leq 1$ is an averaging constant, typically set to a small value such as 0.05, and y (*imgY*) is a new observation at time t . This is more generally called a running average.

cvRunningAvgMask

Evaluates weighted sum of two images, using mask.

```
void cvRunningAvgMask( IplImage* imgY, IplImage* imgU, IplImage* mask, double  
    alpha );
```

<i>imgY</i>	Input image.
<i>imgU</i>	Running average image.
<i>mask</i>	Mask image.
<i>alpha</i>	Weight of input image.

Discussion

The function calculates the weighted sum of pixel values of the input image *imgY* and the running average image *imgU* according to the equation

$$imgU_{n+1} = (1 - alpha) \times imgU_n + alpha \times imgY,$$

if the corresponding pixel values in the mask are nonzero, or does nothing otherwise.

Distance Transform

10

This chapter describes the distance transform functions group.

Reference

cvDistTransform

Calculates distance to closest zero pixel for all nonzero pixels of source image.

```
void cvDistTransform ( IplImage* src, IplImage* dst, CvDistType distType  
    CvDisMaskType maskType float* mask);
```

<i>src</i>	Source image.
<i>dst</i>	Output image (contains calculated distances).
<i>distType</i>	Type of distance, can be CV_DIST_L1, CV_DIST_L2, CV_DIST_C, or CV_DIST_USER.
<i>maskType</i>	Size of distance transform mask, can be CV_DIST_MASK_3x3 or CV_DIST_MASK_5x5.
<i>mask</i>	Pointer to the user-defined mask (used with the distance type CV_DIST_USER).

Discussion

The distance transform function approximates the actual distance from the closest zero pixel with a sum of fixed distance values: two for 3x3 mask and three for 5x5 mask. The result of the distance transform on a 7x7 image with zero central pixel is shown on figure 10-1.

Figure 10-1 3x3 mask

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	0	1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

This example corresponds to a 3x3 mask; in case of user-defined distance type the user sets the distance between two pixels that share the edge and the distance between the pixels that share the corner only. For this case the values are 1 and 1.5 correspondingly. Figure 10-2 shows the distance transform for the same image, but for a 5x5 mask. For the 5x5 mask the user sets the additional distance that is the distance between pixels corresponding to the chess knight move. For this example the additional distance is equal to 2. For `CV_DIST_L1`, `CV_DIST_L2`, and `CV_DIST_C` the optimal precalculated distance values are used.

Figure 10-2 5x5 mask

4.5	3.5	3	3	3	3.5	4
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Threshold Functions

11

This chapter describes threshold functions group.

Reference

cvAdaptiveThreshold

Provides adaptive thresholding binary image.

```
void cvAdaptiveThreshold( IplImage* src, IplImage* dst, double max,  
    CvAdaptiveThreshMethod method, CvThreshType type, double* parameters);
```

<i>src</i>	Source image.
<i>dst</i>	Destination image.
<i>thresh</i>	Threshold parameter.
<i>max</i>	Max parameter, used only with types CV_THRESH_BINARY and CV_THRESH_BINARY_INV.
<i>method</i>	Method for the adaptive threshold definition (now CV_STDDEF_ADAPTIVE_THRESH only).
<i>type</i>	Thresholding type, must be one of <ul style="list-style-type: none">• CV_THRESH_BINARY, <i>val</i> = (<i>val</i> > <i>Thresh</i>?MAX:0);• CV_THRESH_BINARY_INV, <i>val</i> = (<i>val</i> > <i>Thresh</i>?0:MAX);• CV_THRESH_TOZERO, <i>val</i> = (<i>val</i> > <i>Thresh</i>?<i>val</i>:0);• CV_THRESH_TOZERO_INV, <i>val</i> = (<i>val</i> > <i>Thresh</i>?0:<i>val</i>).

parameters Pointer to the input parameters (for the CV_STDDEF_ADAPTIVE_THRESH method *parameters*[0] is size of the neighborhood thresholding, (one of the 1-(3x3), 2-(5x5), or 3-(7x7)), *parameters*[1] is the value of the minimum variance.

Discussion

The function calculates the adaptive threshold for every input image pixel and segments image. The algorithm is as follows. Let $\{f_{ij}\}, 1 \leq i \leq I, 1 \leq j \leq J$ be the input image. For every pixel i, j we calculate the mean m_{ij} and variance v_{ij} given by:

$$m_{ij} = 1/2p \cdot \sum_{s=-p}^p \sum_{t=-p}^p f_{i+s, j+t}, v_{ij} = 1/2p \cdot \sum_{s=-p}^p \sum_{t=-p}^p |f_{i+s, j+t} - m_{ij}|, \text{ where}$$

where $p \times p$ is the neighborhood.

Local threshold for pixel i, j is $t_{ij} = m_{ij} + v_{ij}$ for the $v_{ij} > v_{min}$ and $t_{ij} = t_{i, j-1}$ for the $v_{ij} \leq v_{min}$, where v_{min} is the minimum variance value. If $j = 1$ then $t_{ij} = t_{i-1, j}$, $t_{11} = t_{i_0 j_0}$, where $v_{i_0 j_0} > v_{min}$ and $v_{ij} \leq v_{min}$ for the $(i < i_0) \vee ((i = i_0) \wedge (j < j_0))$.

Output segmented image is calculated as in the function `cvThreshold`.

cvThreshold

Thresholds binary image.

```
void cvThreshold( IplImage* src, IplImage* dst, float thresh, float max
CvThreshType type);
```

<i>src</i>	Source image.
<i>dst</i>	Destination image, could be the same as the parameter <i>src</i> .
<i>thresh</i>	Threshold parameter.
<i>max</i>	Max parameter, used only with types CV_THRESH_BINARY and CV_THRESH_BINARY_INV (see below).
<i>type</i>	Thresholding type, must be one of

- `CV_THRESH_BINARY, val = (val > Thresh MAX:0);`
- `CV_THRESH_BINARY_INV, val = (val > Thresh 0:MAX);`
- `CV_THRESH_TRUNC, val = (val > Thresh Thresh:MAX);`
- `CV_THRESH_TOZERO, val = (val > Thresh val:0);`
- `CV_THRESH_TOZERO_INV, val = (val > Thresh 0:val);`

Discussion

This chapter describes functions for an image area flood filling.

Reference

cvFloodFill, cvFloodFill8

Make a flood filling of connected area of the image

```
void cvFloodFill ( IplImage* img, CvPoint seed_point, double new_val, double  
                  lo_diff, double up_diff, CvConnectedComp* comp );
```

```
void cvFloodFill8( IplImage* img, CvPoint seed_point, double new_val, double  
                  lo_diff, double up_diff, CvConnectedComp* comp );
```

<i>img</i>	Input image which is "repainted" during the function action.
<i>seed_point</i>	Coordinates of the seed point inside the image ROI.
<i>new_val</i>	New value of repainted area pixels.
<i>lo_diff,</i>	Maximal lower difference of the values of appurtenant to repainted area pixel and one of its neighbors.
<i>up_diff</i>	Maximal upper difference of the values of appurtenant to repainted area pixel and one of its neighbors.
<i>comp</i>	Pointer to the connected component structure of the repainted area.

Discussion

The functions fill the seed pixel environs inside which all pixel values are not far from each other. The pixel is considered to be appurtenant to the repainted area if its value v accords to following conditions:

$$v_0 - d_{lw} \leq v \leq v_0 + d_{up}$$

where v_0 is the value of (at least) one of the current pixel neighbors, which is already appurtenant to the repainted area. The first function checks 4-connected environs of each pixel (i.e., its side neighbors), the second function checks 8-connected environs (corner neighbors as well).

This chapter describes Camera Calibration algorithm realization functions.

Overview

Camera parameters

Camera calibration functions are used for calculating intrinsic and extrinsic camera parameters.

Camera parameters are the numbers describing a particular camera configuration. The *intrinsic* camera parameters are those that specify the camera characteristics proper; these parameters include the focal length (the distance between the camera lens and the imaging plane), the location of the image center in pixel coordinates, the effective pixel size, and the radial distortion coefficient of the lens. The *extrinsic* camera parameters describe the spatial relationship between the camera and the world; they are the rotation matrix and translation vector specifying the transformation between the camera and world reference frames.

A camera is modeled by the usual pinhole: the relationship between a 3D point M and its image projection m is given by the formula

$$m = A[Rt]M,$$

where A is the camera intrinsic matrix;

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where (c_x, c_y) are coordinates of the principal point;

(f_x, f_y) are the focal lengths by the axes x and y ;

(R, t) are extrinsic parameters, the rotation matrix R and translate vector t that relates the world coordinate system to the camera coordinate system

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, t = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

Camera usually exhibits significant lens distortion, especially radial distortion. The distortion has 4 coefficients: $k1, k2, k3, k4$.

Use `cvUndistort` to correct the camera lens distortion (fig. 9.2). FIG #####

Algorithm used inside camera calibration.

1. Find homography for all point on series of images;
2. Initialize intrinsic parameters. Distortion set to 0;
3. For which image of pattern find extrinsic parameters;
4. Make main optimization by minimizing error of projection points with all parameters.

Homography

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \text{ is the matrix of homography.}$$

Without a loss of generality, we assume the model plane is on $z = 0$ of the world coordinate system. Let's denote the i^{th} column of the rotation matrix R by r_i . We have

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A[r_1 \ r_2 \ r_3 \ t] \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = A[r_1 \ r_2 \ t] \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

By abuse of notation, we still use M to denote a point the model plane, but $M = [X, Y]^T$, since Z is always equals to 0. In turn, $\tilde{M} = [X, Y, 1]^T$. Therefore, a model point M and its image m are related by the homography H

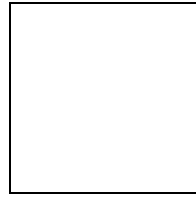
$$s\tilde{m} = H\tilde{M} \text{ with } H = A[r_1 \ r_2 \ t]$$

As is clear, the 3x3 matrix H is defined up to a scalar factor.

Pattern

Calibration may be made using pattern (fig.9.1) FIG####. Pattern has black and white squares on white background. The geometry of pattern must be known. The pattern may be printed using high-quality printer and put on a glass.

Figure 13-1 Pattern



Pattern.

Lens distortion

Camera usually exhibits significant lens distortion, especially radial distortion. The distortion has 4 coefficients k_1, k_2, k_3, k_4 .

Let (u, v) be true pixel image coordinates, i.e., coordinates with ideal projection, and (\tilde{u}, \tilde{v}) be corresponding real observed (distorted) image coordinates. Similarly, (x, y) are ideal (distortion-free) and (\tilde{x}, \tilde{y}) are real (distorted) image physical coordinates.

Taking in account two expansion terms, we have:

$$\begin{aligned}\tilde{x} &= x + x[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \\ \tilde{y} &= y + y[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2]\end{aligned}$$

where k_1 and k_2 are the coefficients of the radial distortion. The center of the radial distortion is the same as the principal point. As $\tilde{u} = u_0 + \alpha \tilde{x}$, and $\tilde{v} = v_0 + \beta \tilde{y}$, where u_0 , v_0 , α , and β are components of the camera intrinsic matrix, we have:

$$\begin{aligned}\tilde{u} &= u + (u - u_0)[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \\ \tilde{v} &= v + (v - v_0)[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2]\end{aligned}$$

The last relations are the basic ones for `cvUndistort` function.

FIGURE

Rotate matrix and vector

Rodrigues conversion (`cvRodrigues`) it is a method to convert rotate vector to rotate matrix or vice versa.

Reference

cvCalibrateCamera

Calibrates camera with single precision.

```
void cvCalibrateCamera( int numImages, int* numPoints, CvSize imageSize,
    CvPoint2D32f* imagePoints32f, CvPoint3D32f* objectPoints32f, CvVect32f
    distortion32f, CvMatr32f cameraMatrix32f, CvVect32f transVects32f,
    CvMatr32f rotMatrs32f, int useIntrinsicGuess);
```

<i>numImages</i>	Number of images.
<i>numPoints</i>	Array of the number of points in each image.
<i>imageSize</i>	Size of image.
<i>imagePoints32f</i>	Pointer to the images.
<i>objectPoints32f</i>	Pointer to the pattern.
<i>distortion32f</i>	Array of four found distortion coefficients.

<i>cameraMatrix32f</i>	Found camera matrix.
<i>transVects32f</i>	Array of translate vectors for each pattern position on the image.
<i>rotMatrs32f</i>	Array of the rotate matrix for each pattern position on the image.
<i>useIntrinsicGuess</i>	Intrinsic guess. If equal to 1, intrinsic guess is needed.

Discussion

The function calculates the camera parameters using information points on the pattern object and pattern object images.

cvCalibrateCamera_64d

Calibrates camera with double precision.

```
void cvCalibrateCamera_64d( int numImages, int* numPoints, CvSize imageSize,
    CvPoint2D64d* imagePoints, CvPoint3D64d* objectPoints, CvVect64d
    distortion, CvMatr64d cameraMatrix, CvVect64d transVects, CvMatr64d
    rotMatrs, int useIntrinsicGuess);
```

<i>numImages</i>	Number of images.
<i>numPoints</i>	Array of the number of points in each image.
<i>imageSize</i>	Size of the image.
<i>imagePoints</i>	Pointer to the images.
<i>objectPoints</i>	Pointer to the pattern.
<i>distortion</i>	Found distortion coefficients.
<i>cameraMatrix</i>	Found camera matrix.
<i>transVects</i>	Array of translate vectors for each pattern position on the image.
<i>rotMatrs</i>	Array of the rotate matrix for each pattern position on the image.

useIntrinsicGuess Intrinsic guess. If equal to 1, intrinsic guess is needed.

Discussion

This function is basically the same as the function `cvCalibrateCamera`, but uses double precision.

cvFindExtrinsicCameraParams

Finds extrinsic camera parameters for pattern.

```
void cvFindExtrinsicCameraParams( int numPoints, CvSize imageSize,
    CvPoint2D32f* imagePoints32f, CvPoint3D32f* objectPoints32f, CvVect32f
    focalLength32f, CvPoint2D32f principalPoint32f, CvVect32f distortion32f,
    CvVect32f rotVect32f, CvVect32f transVect32f);
```

<i>NumPoints</i>	Number of points.
<i>ImageSize</i>	Size of image.
<i>imagePoints32f</i>	Pointer to the image.
<i>objectPoints32f</i>	Pointer to the pattern.
<i>focalLength32f</i>	Focal length.
<i>principalPoint32f</i>	Principal point.
<i>distortion32f</i>	Distortion.
<i>rotVect32f</i>	Rotate vector.
<i>transVect32f</i>	Translate vector.

Discussion

The function finds the extrinsic parameters for pattern.

cvFindExtrinsicCameraParams_64d

Finds extrinsic camera parameters.

```
void cvFindExtrinsicCameraParams_64d( int numPoints, CvSize imageSize,
    CvPoint2D64d* imagePoints, CvPoint3D64d* objectPoints, CvVect64d
    focalLength, CvPoint2D64d principalPoint, CvVect64d distortion, CvVect64d
    rotVect, CvVect64d transVect);
```

<i>NumPoints</i>	Number of points.
<i>ImageSize</i>	Size of image.
<i>imagePoints</i>	Pointer to the image.
<i>objectPoints</i>	Pointer to the pattern.
<i>focalLength</i>	Focal length.
<i>principalPoint</i>	Principal point.
<i>distortion</i>	Distortion.
<i>rotVect</i>	Rotate vector.
<i>transVect</i>	Translate vector.

Discussion

The function finds the extrinsic parameters for pattern in double precision.

cvRodrigues

Converts rotate matrix to rotate vector and vice versa with single precision.

```
void cvRodrigues( CvMatr32f rotMatr32f, CvVect32f rotVect32f, CvMatr32f
    Jacobian32f, CvRodriguesType convType);
```

<i>rotMatr32f</i>	Rotate matrix.
<i>rotVect32f</i>	Rotate vector.

<i>Jacobian32f</i>	Jacobian matrix 3x9.
<i>ConvType</i>	Type of conversion. Must be CV_RODRIGUES_M2V for the converting the matrix to the vector or CV_RODRIGUES_V2M for the converting the vector to the matrix).

Discussion

The function converts the rotation matrix to the rotation vector or vice versa.

cvRodrigues_64d

Converts rotate matrix to rotate vector and vice verse with double precision.

```
void cvRodrigues_64d( CvMatr64d  rotMatr, CvVect64d rotVect, CvMatr64d
Jacobian, CvRodriguesType convType);
```

<i>rotMatr</i>	Rotate matrix.
<i>rotVect</i>	Rotate vector.
<i>Jacobian</i>	Jacobian matrix 3x9.
<i>ConvType</i>	Type of conversion must be CV_RODRIGUES_M2V for converting the matrix to the vector or CV_RODRIGUES_V2M for converting the vector to the matrix).

Discussion

The function converts the rotation matrix to the rotation vector or vice versa.

cvUndistort

Corrects the camera lens distortion

```
void cvUndistort ( IplImage* srcImage, IplImage* dstImage, float* intrMatrix,
float* distCoeffs, int interToggle );
```

<i>srcImage</i>	Source (distorted) image.
<i>dstImage</i>	Destination (corrected) image.
<i>intrMatrix</i>	Matrix of the camera intrinsic parameters.
<i>distCoeffs</i>	Vector of the distortion coefficients.
<i>interToggle</i>	Interpolation toggle.

Discussion

This function corrects camera lens distortion

If *interToggle* = 0, then inter-pixel interpolation is disabled; otherwise bilinear interpolation is used. In the latter case the function acts slower, but quality of the corrected image increases.

Matrix of the camera intrinsic parameters and distortion coefficients k_1, k_2 should be calculated by the function `cvCalibrateCamera`.

cvFindChessBoardCornerGuesses

Finds approximate positions of internal corners of the chessboard.

```
void cvFindChessBoardCornerGuesses(IplImage* img, IplImage* thresh, CvSize
etalon_size, CvPoint2D32f* corners, int *corner_count );
```

<i>img</i>	Source chessboard view. Must be byte-depth, single channel image
<i>thresh</i>	Temporary image of the same size and format as the source image.

<i>etalon_size</i>	Number of squares (both black and white) per the chessboard row and column.
<i>corners</i>	Pointer to the found corner array.
<i>corner_count</i>	Signed value whose absolute value is a number of found corners. A positive number means that a whole chessboard has been found and a negative number means that not all the corners have been found.

Discussion

The function tries to determine whether the input image is a chessboard view and locate internal chessboard corners. If a positive value is returned, all corners have been found and they are already in a certain order (row by row, left to right in every row). For example, a simple chessboard has 8x8 squares and 7x7 internal corners (points, where the squares are tangent). The word “approximate” in the above description means that the found corner coordinates may differ from the actual coordinates by a couple of pixels. To get more precise coordinates, the user may use `cvFindCornersSubP`.

This chapter describes some specific functions for View Morphing algorithm.

Overview

The View Morphing algorithm used to get image from a virtual camera that takes data from two real cameras and some correspondence information besides. The virtual camera could be positioned between the two real cameras.

This part addresses the problem of synthesizing images of real scenes under three-dimensional transformation in viewpoint and appearance. Solving this problem enables interactive viewing of remote scenes on a computer, in which a user can move the virtual camera through the environment. The point to make here is that a three-dimensional scene transformation can be rendered on a video display device by applying simple transformation to a set of basis images of the scene. The virtue of these transformations is that they operate directly on the image and recover only the scene information that is required to accomplish the desired effect. Consequently, the transformations are applicable in a situation where accurate three-dimensional models are difficult or impossible to obtain.

A central topic is the problem of view synthesis, i.e., rendering images of a real scene from different camera viewpoints by processing a set of basis images.

Algorithm

1. Find fundamental matrix. For example using correspondence points on images.
2. Find scanlines for each images.
3. Warp images across scanlines.
4. Find correspondence of warped images.

5. Morph warped images across position of virtual camera.
6. Unwarp image.
7. Delete moire on resulting image.

FIGURE Original images

FIGURE Correspondence points

FIGURE Scan lines

FIGURE Moire on morphed image

FIGURE Resulting morphed image

Morphed image from virtual camera with deleted moire.

Using functions for View Morphing algorithm

1. Find the fundamental matrix using the correspondence points on the two images of cameras by calling the function `cvFindFundamentalMatrix`;
2. Find the number of scanlines in the images for the given fundamental matrix by calling the function `cvFindFundamentalMatrix` with null pointers to the scanlines;
3. Allocate enough memory for:
 - scanlines in the first image, scanlines in the second image, scanlines in the virtual image (for each `numscan*2*4*sizeof(int)`);
 - lengths of scanlines in the first image, lengths of scanlines in the second image, lengths of scanlines in the virtual image (for each `numscan*2*4*sizeof(int)`);
 - buffer for the prewarp first image, the second image, the virtual image (for each `width*height*2*sizeof(int)`);
 - data runs for the first and second images (for each `width*height*4*sizeof(int)`);
 - correspondence data for the first image and the second image (for each `width*height*2*sizeof(int)`);
 - numbers of lines for the first and second images (for each `width*height*4*sizeof(int)`).

4. Find scanlines coordinates by calling the function `cvFindFundamentalMatrix`;
5. Prewarp the first and second images using scanlines data by calling the function `ippiPreWarpImage`;
6. Find runs on the first and second images scanlines by calling the function `cvFindRuns`;
7. Find correspondence information by calling the function `cvDynamicCorrespondMulti`;
8. Find coordinates of scanlines on the virtual image for the virtual camera position `alpha` by calling the function `cvMakeAlphaScanlines`;
9. Morph the prewarp virtual image from the first and second images using correspondence information by calling the function `cvMorphEpilinesMulti`;
10. Postwarp the virtual image by calling the function `cvPostWarpImage`;
11. Delete moire from the resulting virtual image by calling the function `cvDeleteMoire`.

Reference

cvFindFundamentalMatrix

Finds fundamental matrix from correspondence pair points in two images.

```
void cvFindFundamentalMatrix( int* points1, int* points2, int numpoints, int
method, CvMatrix3* matrix);
```

<i>points1</i>	Pointer to the array of correspondence points on the first image.
<i>points2</i>	Pointer to the array of correspondence points on the second image.
<i>numpoints</i>	Number of point pairs.
<i>method</i>	Method for finding the fundamental matrix. Currently not used, must be zero.

matrix Resulting fundamental matrix.

Discussion

The function finds the fundamental matrix from correspondence pair points in two images. If the number of points is too small or the point positions are not good, i.e., lie very close or on the same planar surface, the matrix will not be found correctly.

cvMakeScanlines

Makes scanlines coordinates for two cameras by fundamental matrix.

```
void cvMakeScanlines( CvMatrix3* matrix, CvSize imgSize, int* scanlines_1,
int* scanlines_2, int* lens_1, int* lens_2, int* numlines);
```

<i>matrix</i>	Fundamental matrix.
<i>imgSize</i>	Size of the image.
<i>scanlines_1</i>	Pointer to the array finding scanlines of the first image.
<i>scanlines_2</i>	Pointer to the array of found scanlines of the second image.
<i>lens_1</i>	Pointer to the array of found lengths (in pixels) of scanlines of the first image.
<i>lens_2</i>	Pointer to the array of found lengths (in pixels) of scanlines of the second image.
<i>numlines</i>	Pointer to the variable to storing number of scanlines.

Discussion

The function finds coordinates of scanlines for two images.

This function returns the number of scanlines. The function does nothing except calculating the number of scanlines if the pointers *scanlines_1* or *scanlines_2* equal to zero.

Memory for all arrays must be allocated before calling this function. Let *numscan* be the number of scanlines. Memory must be allocated for:

1. Scanlines in the first image, scanlines in the second image, scanlines in the virtual image (for each $numscan * 2 * 4 * sizeof(int)$);
2. Lengths of scanlines in the first image, lengths of scanlines in the second image, lengths of scanlines in the virtual image (for each $numscan * 2 * 4 * sizeof(int)$);
3. Buffer for the prewarp first image, second image, virtual images (for each $width * height * 2 * sizeof(int)$);
4. Data runs for the first and second images (for each $width * height * 4 * sizeof(int)$);
5. Correspondence data for the first image and the second image (for each $width * height * 2 * sizeof(int)$).

cvPreWarpImage

Finds prewarp of given image.

```
void cvPreWarpImage( int numLines, IplImage* img, uchar* dst, int* dst_nums,
                    int* scanlines);
```

<i>numLines</i>	Number of scanlines for the image.
<i>img</i>	Image to prewarp.
<i>dst</i>	Data to store for the prewarp image.
<i>dst_nums</i>	Pointer to the array of lengths of scanlines.
<i>scanlines</i>	Pointer to the array of coordinates of scanlines.

Discussion

The function finds prewarp of the given image across scanlines. Memory must be allocated before calling this function. Memory size is $\max(width, height) * numscanlines * size(char) * 3$.

cvFindRuns

Finds runs in the two prewarp images.

```
void cvFindRuns( int numLines, uchar* prewarp_1, uchar* prewarp_2, int*
    line_lens_1, int* line_lens_2, int* runs_1, int* runs_2, int* num_runs_1,
    int* num_runs_2);
```

<i>numLines</i>	Number of scanlines.
<i>prewarp_1</i>	Prewarp data of the first image.
<i>prewarp_2</i>	Prewarp data of the second image.
<i>line_lens_1</i>	Array of lengths of scanlines on the first image.
<i>line_lens_2</i>	Array of lengths of scanlines on the second image.
<i>runs_1</i>	Array of runs in each line on the first imageruns data 1.
<i>runs_2</i>	Array of runs in each line on the second imageruns data 1.
<i>num_runs_1</i>	Array of numbers of runs in each line on the first image.
<i>num_runs_2</i>	Array of numbers of runs in each line on the second image.

Discussion

The function finds runs in the two prewarp images. Memory must be allocated before calling this function. Memory size for one array of runs is

```
max(width,height)*numscanlines*3*sizeof(int).
```

cvDynamicCorrespondMulti

Finds correspondence between two sets of runs of two warped images.

```
cvDynamicCorrespondMulti( int lines, int* first, int* first_runs, int*
    second, int* second_runs, int* first_corr, int* second_corr);
```

<i>lines</i>	Number of scanlines.
--------------	----------------------

<i>first</i>	Array of runs of the first image.
<i>first_runs</i>	Array of numbers of runs in each scanline of the first image.
<i>second</i>	Array of runs of the second image.
<i>second_runs</i>	Array of numbers of runs in each scanline of the second image.
<i>first_corr</i>	Array of find correspondence information for the first image.
<i>second_corr</i>	Array of find correspondence information for the second image.

Discussion

The function finds correspondence between two sets of runs of two images. The function finds runs in the two prewarp images. Memory must be allocated before calling this function. Memory size for one array of correspondence information is `max(width,height)*numscanlines*3*sizeof(int)`.

cvMakeAlphaScanlines

Finds coordinates of scanlines for image for virtual camera position.

```
void cvMakeAlphaScanlines( int* scanlines_1, int* scanlines_2, int*
    scanlines_a, int* lens, int numlines, float alpha);
```

<i>scanlines_1</i>	Pointer to the array of first scanlines.
<i>scanlines_2</i>	Pointer to the array of second scanlines.
<i>scanlines_a</i>	Pointer to the array of found scanlines on the virtual image.
<i>lens</i>	Pointer to the array of lengths of found scanlines on virtual image.
<i>numlines</i>	Number of scanlines.
<i>alpha</i>	Position of virtual camera (0.0 - 1.0).

Discussion

The function finds coordinates of scanlines for the virtual camera with the given camera position.

Memory must be allocated before calling this function. Memory size for the array of correspondence runs is $\text{numscanlines} * 2 * 4 * \text{sizeof}(\text{int})$. Memory size for the array of the scanline length is $\text{numscanlines} * 2 * 4 * \text{sizeof}(\text{int})$.

cvMorphEpilinesMulti

Morphs two prewarp images using corresponding information.

```
void cvMorphEpilinesMulti( int lines, uchar* first_pix, int* first_num, uchar*
    second_pix, int* second_num, uchar* dst_pix, int* dst_num, float alpha,
    int* first, int* first_runs, int* second, int* second_runs, int*
    first_corr, int* second_corr);
```

<i>lines</i>	Number of scanlines in the prewarp image.
<i>first_pix</i>	Pointer to the first prewarp image.
<i>first_num</i>	Pointer to the array of numbers of points in each scanline in the first image.
<i>second_pix</i>	Pointer to the second prewarp image.
<i>second_num</i>	Pointer to the array of numbers of points in each scanline in the second image.
<i>dst_pix</i>	Pointer to the resulting morphed warped image.
<i>dst_num</i>	Pointer to the array of numbers of points in each line.
<i>alpha</i>	Virtual camera position (0.0 - 1.0).
<i>first</i>	First sequence of runs.
<i>first_runs</i>	Pointer to the number of runs in each scanline on the first image.
<i>second</i>	Second sequence of runs.
<i>second_runs</i>	Pointer to the number of runs in each scanline on the second image.
<i>first_corr</i>	Pointer to the array of found correspondence information for the first run.

second_corr Pointer to the array of found correspondence information for the second runs

Discussion

The function morphs two prewarp images using corresponding information.

cvPostWarpImage

Finds postwarp for given image data.

```
void cvPostWarpImage( int numLines, uchar* src, int* src_nums, IplImage* img,
    int* scanlines);
```

numLines Number of scanlines.
src Pointer to the prewarp image virtual image.
src_nums Number of scanlines on the image.
img Resulting unwarp image.
scanlines Pointer to the array of scanlines data.

Discussion

The function finds postwarp for the given image data.

cvDeleteMoire

Deletes moire on the given image.

```
void cvDeleteMoire( IplImage* img);
```

img Image.

Discussion

The post-morphing post-warped image has got black points: the postwarped image is created by lines, which means that every point may not be filled. The function deletes moire (black points) on the given image by the color of neighbor points. If all scanlines are horizontal, this function may be not used.

Motion Templates

15

This chapter describes Motion Templates functions.

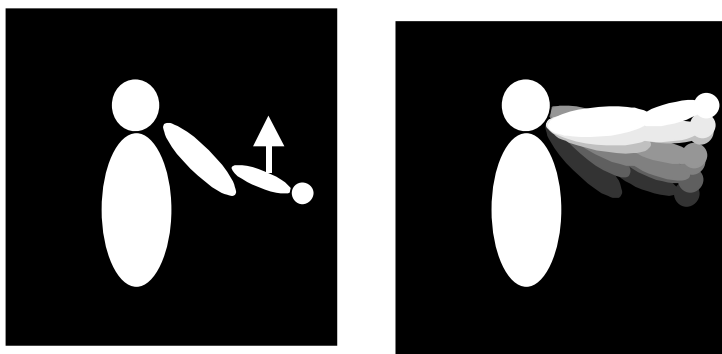
Overview

The functions described in this section are designed to generate motion template images that can be used to rapidly determine where a motion occurred, how it occurred, and in which direction it occurred. The algorithms are based on papers by Davis and Bobick and Davis and Bradski. These functions operate on images that are the output of frame or background differencing, or other image segmentation operations; thus the input and output image types will all be grayscale (one color channel). The pixel types could be 8U, 8S or 32F.

Motion representation and normal optical flow method

Motion representation

We capture a foreground silhouette of the moving object or person as shown in Figure 15-1 below (left). As the person or object moves, we create a “layered history” of the resulting motion by copying the most recent foreground silhouette as the highest values in the motion history image; typically this “highest value” is just a floating point timestamp of time since the code has been running in milliseconds. The result is shown in Figure 15-1 (right) which we call the Motion History Image (MHI). A pixel level or a time delta threshold (as appropriate) is set such that pixel values in the MHI image that fall below that threshold are set to zero.

Figure 15-1 Motion History Image: how the motion took place

The most recent motion has the highest value, earlier motions have decreasing values subject to a threshold below which the value is set to zero. Below the different stages of motion templates creating and processing are described.

A) Updating MHI images

Generally, we work with floating point images since we read system time differences in milliseconds from application launch time, convert the time differences into a floating point number and use that number as the value of our most recent silhouette. We write this current silhouette over the past silhouettes and threshold away pixels that are too old (past a maximum mhi_duration) to create the Motion History Image (MHI).

B) Making the motion gradient image

1. We start with the MHI image as shown in Figure 15-1(left).
2. We apply x and y 3x3 Sobel operators to the image.
3. If the resulting response at a pixel location (x,y) is $S_x(x, y)$ to the x Sobel operator and $S_y(x, y)$ to the y operator, then the orientation of the gradient is calculated as:

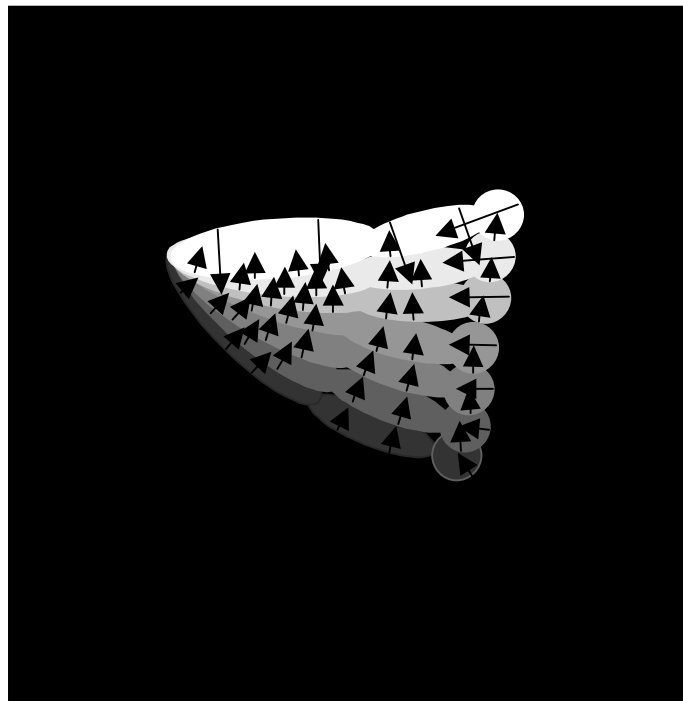
$$A(x, y) = \arctan(S_y((x, y)/S_x(x, y)), \quad (15)$$

and the magnitude of the gradient is:

$$M(x, y) = \sqrt{S_x^2(x, y) + S_y^2(x, y)} \quad (16)$$

4. The equations are applied to the image yielding the direction or angle of flow image superimposed (just for reference) over the MHI image as shown in Figure 15-2.

Figure 15-2 Direction of flow image: Applying the X and Y Sobel operator to the MHI and using the above equations gives the angle and magnitude of flow as shown here superimposed over the MHI.



Note that the outer boundary pixels give incorrect motion angles.

5. Note that in Figure 15-2, the boundary pixels of the MH region give incorrect motion angles and magnitudes. To correct for this we threshold away magnitudes that are either too large or too small yielding the results shown in Figure 15-3.

Figure 15-3 Resulting normal motion directions after thresholding for magnitudes that are too large or small.

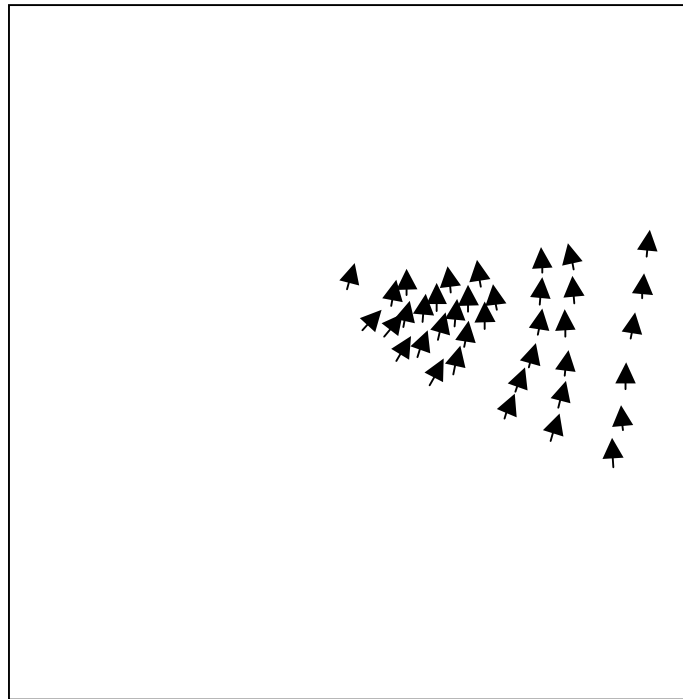
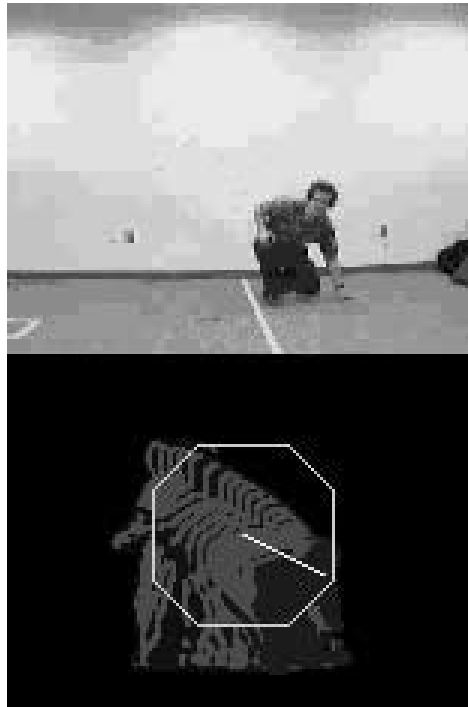


Figure 15-4 shows the output of the motion gradient function described in the section above together with the marked direction of motion flow.

C) Finding Regional Orientation or Normal Optical Flow

Figure 15-4 MHI image of a person kneeling



The current silhouette is in bright blue with past motions in dimmer and dimmer blue. Red lines show where valid normal flow gradients were found. The white line shows computed direction of global motion weighted towards the most recent direction of motion.

To get the most recent, salient global motion, we take a histogram of the motions resulting from processing (see Figure 15-3). We then need to find the average orientation of a circular function: angle in degrees. We do this by first finding the maximal peak in the orientation histogram, and then finding the average of minimum differences from a this base angle. Orientations are weighted by recency.

Motion Segmentation

Usually, it is not necessary to calculate the motion orientation for the whole image. So we want to group motion regions that were produced by the movement of parts or the whole of the object of interest. Labeling motion regions connected to the current silhouette using a downward stepping floodfill enable us to identify areas of motion directly attached to parts of the object of interest.

By construction of tMHI image, the most recent silhouette has the maximal values (e.g., most recent timestamp) in it. We scan the image until we find this value, then walk along the silhouette's contour to find attached areas of motion. The algorithm of creating masks to segment motion region is as follows

1. Scan the tMHI until we find a pixel of the current timestamp (most recent silhouette), mark that region by a floodfill;
2. Walk around the boundary of the current silhouette region looking outside for recent (within a threshold) unmarked motion history "steps". When a suitable step is found, mark it with downward floodfill. If the size of the fill is not big enough, zero out the area.
3. [Optional]:
 - Within each downfill, record locations of minimums (or record locations of predetermined values);
 - Perform separate floodfills up from each location found in;
 - Combine separately (by logical AND) each upfill with downfill it belonged to.
4. Store the segmented motion mask that were found.
5. Continue the boundary "walk" until the silhouette has been circumnavigated.
6. [Optional] Go to 1 until all current silhouette regions are found.

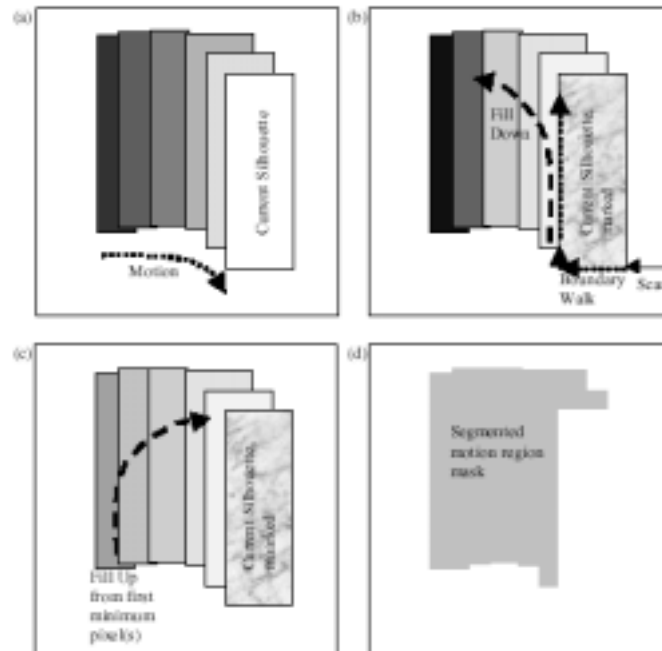


Figure 6 (a) tMHI from a moving “block”. (b) Find current silhouette region, mark it; “walk” the boundary and fill downwards wherever a step found; [Optionally:] keep location(s) of minimums. (c) Fill upwards from the minimums. (d) Combine the down fill and up fill region(s) separately. Store the found motion masks. Search for other silhouette regions.

The functions that do all of the above are described below.

Reference

cvUpdateMHIByTime

Updates motion history image with silhouette of floating point current system time.

```
void cvUpdateMHIByTime(IplImage* silhouette, IplImage* mhi, double timestamp,
    double mhi_duration);
```

<i>silhouette</i>	Silhouette image that has non-zero pixels where the motion occurs.
<i>mhi</i>	Motion history image, both an input and output parameter.
<i>timestamp</i>	Floating point current time in the format: <code>seconds.milliseconds</code> .
<i>mhi_duration</i>	MHI motions older than this threshold will be deleted.

Discussion

The function updates the motion history image, putting the current *timestamp* value to those *mhi* pixels that have non-zero corresponding silhouette pixels. The function also removes *mhi* pixels older than *timestamp* - *mhi_duration* if the corresponding silhouette values are 0.

cvCalcMotionGradient

Calculates gradient orientation of motion history image.

```
void cvCalcMotionGradient( IplImage* mhi, IplImage* mask, IplImage*
    orientation, int aperture_size, double maxTDelta, double minTDelta );
```

<i>mhi</i>	Motion history image.
<i>mask</i>	Mask image. Marks pixels where motion gradient data is correct. Output parameter.

<i>orientation</i>	Motion gradient orientation image. Contains angles from 0 to ~360 degrees.
<i>aperture_size</i>	Size of aperture used to calculate derivatives. Value should be odd, e.g., 3, 5, etc.
<i>maxTDelta</i>	Consider the gradient orientation valid if the difference between the maximum and minimum <i>mhi</i> values within a pixel neighborhood is lower than this threshold.
<i>minTDelta</i>	Consider the gradient orientation valid if the difference between the maximum and minimum <i>mhi</i> values within a pixel neighborhood is greater than this threshold.

Discussion

The function calculates the derivatives D_x and D_y for the *mhi* image and then calculates orientation of the gradient using the formula

$$\phi = \begin{cases} 0, & x = 0, y = 0 \\ \arctan(y/x) & \text{else} \end{cases}$$

Finally, the function masks off pixels with a very small (less than *minTDelta*) or very large (greater than *maxTDelta*) difference between the minimum and maximum *mhi* values in their neighborhood. The neighborhood for finding the minimum and maximum has the same size as derivative kernels.

cvCalcGlobalOrientation

Calculates global motion orientation of some selected region.

```
void cvCalcGlobalOrientation( IplImage* orientation, IplImage* mask, IplImage*
    mhi, double curr_mhi_timestamp, double mhi_duration );
```

orientation Orientation image.

<i>mask</i>	Mask image. Marks pixels where motion gradient data are valid and should be used to calculate global orientation.
<i>mhi</i>	Motion history image.
<i>curr_mhi_timestamp</i>	Current time in <code>seconds.milliseconds</code> .
<i>mhi_duration</i>	Maximal duration of motion track in <code>seconds.milliseconds</code> .

Discussion

The function calculates the general motion direction in the selected region.

At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all orientation vectors (a more recent motion - greater weights). The resultant angle is `<basic orientation> + <shift>`.

cvSegmentMotion

Finds segments of most recent motion on image.

```
void cvSegmentMotion( IplImage* mhi, IplImage* seg_mask, CvMemStorage*
    storage, CvSeq** components, double timestamp, double seg_thresh );
```

<i>mhi</i>	Motion history image.
<i>seg_mask</i>	Image were mask found to be stored.
<i>Storage</i>	Pointer to the memory storage, where sequence of components should be saved.
<i>components</i>	Sequence of components found by the function.
<i>timestamp</i>	Floating point current time in the format: <code>seconds.milliseconds</code> .
<i>seg_thresh</i>	Segmentation threshold. (Recommended to be equal to the interval between motion history “steps” or greater).

Discussion

The function finds the motion segments attached to the pixels when the most recent motion occurs and fills the segments with values (1,2,3...).

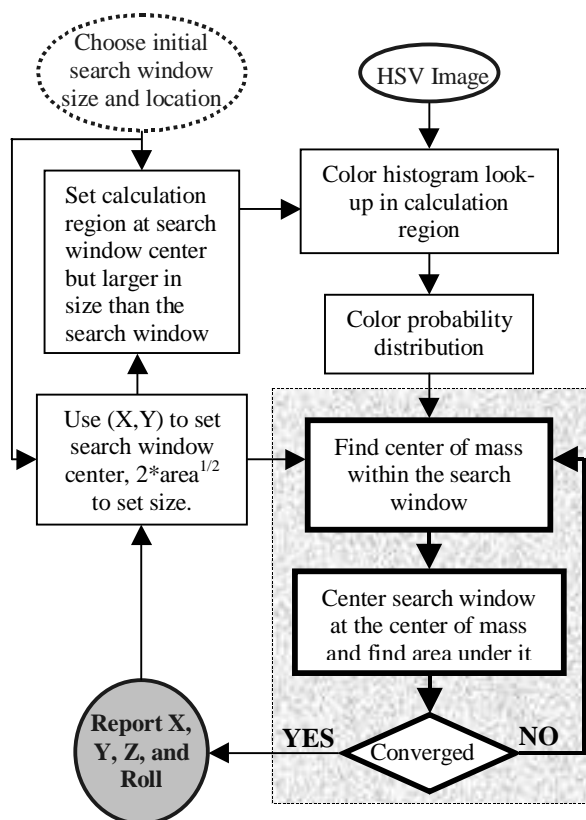
Stores the properties of these segments (rectangle contained the segment, area, value) in the sequence of components.

This chapter describes CamShift algorithm realization functions.

Overview

CamShift stands for the “Continuously Adaptive Mean-SHIFT” algorithm. Figure 16-1 summarizes the algorithm described below. For each video frame, the raw image is converted to a color probability distribution image via a color histogram model of the color being tracked (flesh for face tracking). The center and size of the color object are found via the CamShift algorithm operating on the color probability image. The current size and location of the tracked object are reported and used to set the size and location of the search window in the next video image. The process is then repeated for continuous tracking. The algorithm is a generalization of the Mean Shift algorithm, highlighted in gray in Figure 16-1.

Figure 16-1 Block diagram of the CamShift algorithm, which is based on the Mean Shift algorithm highlighted in gray



CamShift operates on a 2D color probability distribution image produced from histogram back-projection (see histogram section, this document). The core part of the CamShift algorithm is the Mean Shift algorithm:

The Mean shift part of the algorithm (gray area in Figure 16-1) is as follows

1. Choose the search window size;
2. Choose the initial location of the search window;
3. Compute the mean location in the search window;

4. Center the search window at the mean location computed in Step 3;
5. Repeat Steps 3 and 4 until convergence (or until the mean location moves less than a preset threshold).

Calculating center of mass of the 2D probability distribution

For discrete 2D image probability distributions, the mean location (the centroid) within the search window (Steps 3 and 4 above) is found as follows

Find the zeroth moment

$$M_{00} = \sum_x \sum_y I(x, y)$$

Find the first moment for x and y

$$M_{10} = \sum_x \sum_y xI(x, y); M_{01} = \sum_x \sum_y yI(x, y)$$

Mean search window location (the centroid) then would be

$$x_c = \frac{M_{10}}{M_{00}}; y_c = \frac{M_{01}}{M_{00}}$$

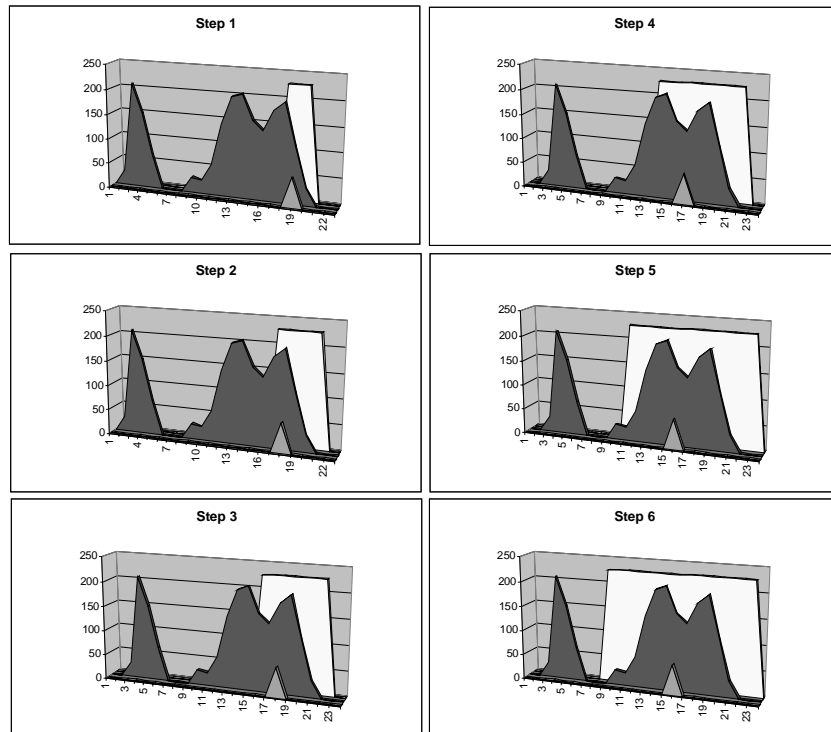
where $I(x, y)$ is the pixel (probability) value in the position (x, y) on the image, and x and y range over the search window.

Unlike the Mean Shift algorithm, which is designed for static distributions, CamShift is designed for dynamically changing distributions. These occur when objects in video sequences are being tracked and the object moves so that the size and location of the probability distribution changes in time. The CamShift algorithm adjusts the search window size in the course of its operation. Initial window size can be set at any reasonable value. For discrete distributions (digital data), the minimum window length or width is three. Instead of a set, or externally adapted window size, CamShift relies on the zeroth moment information, extracted as part of the internal workings of the algorithm, to continuously adapt its window size within or over each video frame.

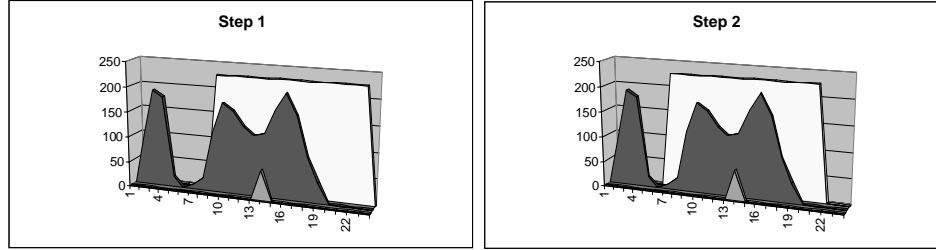
How to calculate the CamShift algorithm

1. First, set the calculation region of the probability distribution to the whole image;
2. Choose the initial location of the 2D mean shift search window;
3. Calculate the color probability distribution in the 2D region centered at the search window location in an ROI slightly larger than the mean shift window size;
4. Mean shift to convergence or for a set number of iterations. Store the zeroth moment (area or size) and mean location;
5. For the next video frame, center the search window at the mean location stored in Step 4 and set the window size to a function of the zeroth moment found there. Go to Step 3.

Figure 16-2 shows CamShift finding the face center on a 1D slice through a face and hand flesh hue distribution. Figure 16-3 shows the next frame when the face and hand flesh hue distribution has moved, convergence is reached in two iterations.

Figure 16-2 Cross section of flesh hue distribution for a face and a hand

Rectangular CamShift window is shown behind the hue distribution, window center is marked by triangle in front. CamShift is shown iterating to convergence down the left then right columns.

Figure 16-3 Next frame, head and hand flesh hue distribution moves

Starting from the converged search location in Figure 16-2 bottom right, CamShift converges on new center of distribution in two iterations.

Calculation of the 2D orientation or Roll of the probability distribution

The 2D orientation of the probability distribution is also easy to obtain by using the second moments during the course of CamShift operation where (x, y) range over the search window, and $I(x, y)$ is the pixel (probability) value at (x, y) :

Second moments are

$$M_{20} = \sum_x \sum_y x^2 I(x, y), \quad M_{02} = \sum_x \sum_y y^2 I(x, y)$$

Then the object orientation (major axis) is

$$\theta = \frac{\arctan \left(\frac{2 \left(\frac{M_{11}}{M_{00}} - x_c y_c \right)}{\left(\frac{M_{20}}{M_{00}} - x_c^2 \right) - \left(\frac{M_{02}}{M_{00}} - y_c^2 \right)} \right)}{2}$$

The first two Eigen values (major length and width) of the probability distribution “blob” found by CamShift may be calculated in closed form as follows. Let

$$a = \frac{M_{20}}{M_{00}} - x_c^2, \quad b = 2 \left(\frac{M_{11}}{M_{00}} - x_c y_c \right), \quad \text{and} \quad c = \frac{M_{02}}{M_{00}} - y_c^2$$

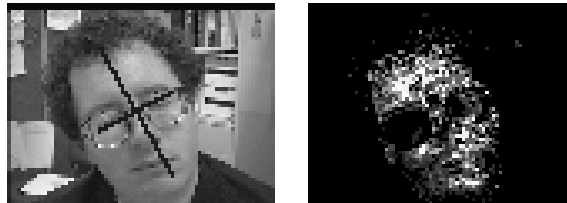
Then length l and width w from the distribution centroid are

$$l = \sqrt{\frac{(a+c) + \sqrt{b^2 + (a-c)^2}}{2}},$$

$$w = \sqrt{\frac{(a+c) - \sqrt{b^2 + (a-c)^2}}{2}}$$

When used in face tracking, the above equations give us head roll, length, and width as marked in Figure 16-4.

Figure 16-4 Orientation of the flesh probability distribution marked on the source video image.



Reference

cvCamShift

Calculates face center, orientation, and size.

```
void cvCamShift( IplImage* imgProb, CvRect windowIn, CvTermCriteria criteria,
                 CvRect* windowOut, float* orientation, float* len, float* width, float*
                 area, int* itersUsed);
```

<i>imgProb</i>	2D object probability distribution.
<i>windowIn</i>	IppiRect of the CamShift Window initial size and location.
<i>criteria</i>	Criteria applied to define the stop of finding the window (see discussion).

<i>windowOut</i>	Location, height, and width of converged CamShift window (output).
<i>orientation</i>	If != NULL, return distribution <i>orientation</i> .
<i>len</i>	If != NULL, return equivalent <i>len</i> .
<i>width</i>	If != NULL, return equivalent <i>width</i> .
<i>area</i>	Sum of all elements in result window (zero th moment).
<i>itersUsed</i>	Returns number of iterations CamShift took to converge.

Discussion

The function iterates face center, orientation and size, and window size until converging to the following criteria:

```
max(newCenter.x - oldCenter.x, newCenter.y - oldCenter.y ) <
criteria.epsilon,
```

or when function does *criteria.maxIters* iterations.

cvMeanShift

Calculates face center.

```
void cvMeanShift( IplImage* imgProb, CvRect windowIn, CvTermCriteria criteria,
CvRect* windowOut, float* area, int* itersUsed);
```

<i>imgProb</i>	2D object probability distribution.
<i>windowIn</i>	IppiRect of search window initial size.
<i>criteria</i>	Criteria applied to define the stop of finding the window (see discussion).
<i>windowOut</i>	Location, height and width of the converged search window.
<i>area</i>	Sum of all elements in the result window (zero th moment).
<i>itersUsed</i>	Returns number of iterations CamShift took to converge.

Discussion

The function iterates face center and window position until converging to the following criteria:

```
Max(newCenter.x - oldCenter.x, newCenter.y - oldCenter.y ) <
criteria.epsilon,
```

or when function does criteria.maxIters iterations.

Active Contours

17

This chapter describes functions for working with active contours (snakes).

Overview

The snake was presented in [Kass88] as an energy-minimizing parametric closed curve guided by external forces. Energy functional associated with the snake $E = E_{int} + E_{ext}$, where E_{int} is the internal energy formed by the snake configuration, E_{ext} is the external energy formed by external forces affecting the snake. The aim of the snake is to find a location that will minimize the energy.

Let p_1, \dots, p_n be a discrete representation of a snake, i.e., a sequence of points on an image plane.

We define $E = E_{cont} + E_{curv}$, where E_{cont} is the contour continuity energy and E_{curv} is the contour curvature energy.

Below follow OpenCV definitions for energy terms.

$E_{cont} = |\bar{d} - \|p_i - p_{i-1}\||$, where \bar{d} is the average distance between all pairs $(p_i - p_{i-1})$.

Minimization of the summary continuity energy over the snake forces the snake points to try to become equidistant.

$$E_{curv} = \|p_{i-1} - 2p_i + p_{i+1}\|^2$$

In [Kass88] external energy was represented as $E_{ext} = E_{img} + E_{con}$.

E_{img} – image energy. E_{con} – energy of additional constraints.

Two variants of image energy are proposed:

$$E_{img} = -I,$$

where I is the image intensity. In this case the snake will be attracted to the bright lines of image.

$E_{img} = -\|grad(I)\|$. The snake will be attracted to image edges.

Variant of external constraint is described in [Kass88]. Imagine the snake points connected by springs with certain points on the image. Spring force $k(x - x_0)$ will produce the energy $\frac{kx^2}{2}$.

These forces will pull snake points to fixed positions, which can be useful when snake points need to be fixed.

OpenCV does not support this opportunity now.

Summary energy at every point can be written as

$$E_i = \alpha_i E_{cont,i} + \beta_i E_{curv,i} + \gamma_i E_{img,i} \quad (1)$$

where α, β, γ are the weights of every kind of energy. The full snake energy is the sum of E_i over all points.

The meanings of α, β, γ are following.

α is responsible for contour continuity, i.e., a big α will make snake points more evenly spaced.

β is responsible for snake corners, i.e., a big β for a certain point will make the angle between snake edges more obtuse.

Lastly, a big γ will make the snake point be more sensitive to the image energy, rather than to continuity or curvature.

Note, that only relative values of α, β, γ in the snake point are relevant.

The following way of working with snakes is proposed:

- create a snake with initial configuration;
- at every point define weights α, β, γ ;
- allow the snake to minimize its energy;
- evaluate the snake position. If required, adjust α, β, γ (maybe image) and repeat the previous step.

There are three well-known algorithms for minimizing snake energy.

In [Kass88] the minimization based on variational calculus.

In [Yuille89] dynamic programming is used

The greedy algorithm is proposed at [Williams92].

The last algorithm is the most efficient and gives a quite good result.

Here is the scheme of this algorithm:

Consequently for each snake point do:

For every location from point's neighborhood compute E , using Equation (1).

Before computing E each energy term E_{cont} , E_{curv} , E_{img} must be normalized using formula $E_{normalized} = (E_{img} - min) / (max - min)$, where max and min are maximal and minimal energy in scanned neighborhood.

Choose location with minimum energy.

Move snakes point to this location.

Repeat 1) until convergence.

Criteria of convergence, which we shall use, is:

number of maximum iterations is achieved;

number of points, moved at last iteration, is quite small.

In [Williams92] the authors proposed way to adjusting b coefficient for corner estimation during minimization process (they call it high-level feedback). We don't do it in our realization, allowing the user to do it.

Reference

cvSnakeImage, cvSnakeImageGrad

Change contour position, minimizing its energy.

```
void cvSnakeImage( IplImage* image, CvPoint* points, int length, float* alpha,
    float* beta, float* gamma, int coeffUsage, CvSize win, CvTermCriteria
    criteria);
```

```
void cvSnakeImageGrad( IplImage* image, CvPoint* points, int length, float*
    alpha, float* beta, float* gamma, int coeffUsage, CvSize win,
    CvTermCriteria criteria);
```

<i>image</i>	Pointer to the source image.
<i>points</i>	Points of the contour.
<i>length</i>	Number of points in the contour.
<i>alpha</i>	Weight of continuity energy.
<i>beta</i>	Weight of curvature energy.
<i>gamma</i>	Weight of image energy.
<i>coeffUsage</i>	Variant of usage of previous three parameters. <ul style="list-style-type: none"> • CV_VALUE uses <i>alpha</i>, <i>beta</i>, <i>gamma</i> as pointers to a single value that will be used for all points; • CV_ARRAY, pointers to arrays with coefficients different for all points of the snake. These array must have the size equal to the snake size.
<i>win</i>	Size of neighborhood of every point used to search the minimum (must be odd).
<i>criteria</i>	Termination criteria.

Discussion

The first function uses image intensity as image energy, second one uses magnitude of image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved to the current iteration.

If the number of moved points is less than the epsilon, the function will terminate.

Overview

Most of known papers devoted to motion estimation use the term “*optical flow*”. Optical flow is defined as an apparent motion of image brightness. Let $I(x, y, t)$ be the brightness of image, which changes in time to provide an image sequence).

Let’s make two main assumptions:

1. brightness $I(x, y, t)$ smoothly depends on coordinates x, y on most of the image;
2. brightness of every point of moving or static object does not change in time.

Let some object in the image (or some point of an object) move and after time dt the object displacement is (dx, dy) . Using Taylor series for brightness $I(x, y, t)$, we have the following:

$$I(x + dx, y + dy, t + dt) = I(x, y, t) + \frac{\partial I}{\partial x} dx + \frac{\partial I}{\partial y} dy + \frac{\partial I}{\partial t} dt + \dots, \quad (1)$$

where ... are higher order terms.

Under assumption 2, we have

$$I(x + dx, y + dy, t + dt) = I(x, y, t), \quad (2)$$

and

$$\frac{\partial I}{\partial x} dx + \frac{\partial I}{\partial y} dy + \frac{\partial I}{\partial t} dt + \dots = 0. \quad (3)$$

Dividing (3) by dt and defining

$$\frac{dx}{dt} = u, \quad \frac{dy}{dt} = v \quad (4)$$

we have an equation

$$-\frac{\partial I}{\partial t} = \frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v, \quad (5)$$

which is usually called optical flow constraint equation. Here u and v are components of optical flow field in x and y coordinates respectively. Since equation (5) has more than one solution, more constraints are required.

Some variants of further steps may be chosen. Below follows a brief overview of the options available.

Lucas & Kanade technique

Using the optical flow equation for group of adjacent pixels and assuming that all of them have the same velocity, we can make a system of linear equations.

In a non-singular system for two pixels we can compute a velocity vector to solve the system. However, combining equations for more than two pixels is more effective. We might get a system that has no solution; yet we can solve it roughly, using the least square method. We will use weighted combination of equations. This method involves the solution of 2×2 linear system.

$$\sum_{x,y} W(x,y) I_x I_y u + \sum_{x,y} W(x,y) I_y^2 v = - \sum_{x,y} W(x,y) I_y I_t,$$

$$\sum_{x,y} W(x,y) I_x^2 u + \sum_{x,y} W(x,y) I_x I_y v = - \sum_{x,y} W(x,y) I_x I_t,$$

where $W(x,y)$ is the Gaussian window. We will represent the Gaussian window as a composition of two separable kernels with binomial coefficients.

Horn & Schunck technique

Horn and Shunk propose a technique that assumes the smoothness of the estimated optical flow field. This constraint can be formulated as

$$S = \iint_{\text{image}} \left[\left(\frac{\partial u}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial y} \right)^2 + \left(\frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial y} \right)^2 \right] dx dy. \quad (6)$$

Our optical flow solution can deviate from the optical flow constraint. To express this deviation we can use following integral:

$$C = \iint_{\text{image}} \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right)^2 dx dy. \quad (7)$$

We must minimize $S + \lambda C$, where λ is a parameter (Lagrangian multiplier). Typically we must take a smaller λ for a noisy image and larger for a quite accurate image.

To minimize $S + \lambda C$, we must solve a system of two second-order differential equations for the whole image:

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= \lambda \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right) \frac{\partial I}{\partial x}, \\ \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} &= \lambda \left(\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t} \right) \frac{\partial I}{\partial y}.\end{aligned}\tag{8}$$

Iterative method could be applied for the purpose when a number of iterations are made for each pixel. This technique for two consecutive images seems to be computationally expensive because of iterations, but for a long sequence of images only an iteration for two images must be done, if the result of previous iteration is chosen as initial approximation.

Block matching

This technique does not use an optical flow equation directly. Let's consider the image tiled with small blocks. For every block of the first image we must choose a block on the second image which have the maximum degree of similarity and nearest coordinates relative to the given one. Cross correlation, squared difference, etc. can be chosen as criteria of similarity

Reference

cvCalcOpticalFlowHS

Calculates optical flow for two images using Horn & Schunck algorithm.

```
void cvCalcOpticalFlowHS( IplImage* srcA, IplImage* srcB, int usePrevious,
    IplImage* velx, IplImage* vely, double lambda, CvTermCriteria criteria);
```

<i>imgA</i>	First image.
<i>imgB</i>	Second image.
<i>usePrevious</i>	Use previous (input) velocity field.

<i>velx</i>	Horizontal component of the optical flow.
<i>vely</i>	Vertical component of the optical flow.
<i>lambda</i>	Lagrangian multiplier.
<i>criteria</i>	Criteria of termination of velocity computing.

Discussion

The function computes flow for every pixel, thus output images must have the same size as input.

cvCalcOpticalFlowLK

Calculates optical flow for two images using Lucas & Kanade technique.

```
void cvCalcOpticalFlowBM( IplImage* srcA, IplImage* srcB, CvSize winSize,  
IplImage* velx, IplImage* vely);
```

<i>imgA</i>	First image.
<i>imgB</i>	Second image.
<i>winSize</i>	Size of the averaging window used for grouping pixels.
<i>velx</i>	Horizontal component of the optical flow.
<i>vely</i>	Vertical component of the optical flow.

Discussion

The function computes flow for every pixel, thus output images must have the same size as input.

cvCalcOpticalFlowBM

Calculates optical flow for two images by block matching method.

```
void cvCalcOpticalFlowBM( IplImage* srcA, IplImage* srcB, CvSize blockSize,  
    CvSize shiftSize, CvSize maxRange, int usePrevious, IplImage* velx,  
    IplImage* vely);
```

<i>imgA</i>	First image.
<i>imgB</i>	Second image.
<i>blockSize</i>	Size of basic blocks that are compared.
<i>shiftSize</i>	Block coordinate increments.
<i>maxRange</i>	Size of the scanned neighborhood in pixels around block.
<i>usePrevious</i>	Use previous (input) velocity field.
<i>velx</i>	Horizontal component of the optical flow.
<i>vely</i>	Vertical component of the optical flow.

Discussion

The function calculates optical flow for two images using the block-matching algorithm. Velocity is computed for every block (not every pixel), so velocity image pixels correspond to input image blocks and the velocity image must have the following size:

$$\text{velocityFrameSize.width} = \left\lceil \frac{\text{imageSize.width}}{\text{blockSize.width}} \right\rceil,$$
$$\text{velocityFrameSize.height} = \left\lceil \frac{\text{imageSize.height}}{\text{blockSize.height}} \right\rceil$$

cvCalcOpticalFlowPyrLK

Calculates optical flow for two images using iterative Lucas-Kanade method in pyramids.

```
void cvCalcOpticalFlowPyrLK(IplImage* imgA, IplImage* imgB, IplImage* pyrA,
    IplImage* pyrB, CvPoint2D32f* featuresA, CvPoint2D32f* featuresB, int
    count, CvSize winSize, int level, char* status, float* error,
    CvTermCriteria criteria, int flags );
```

<i>imgA</i>	First frame (time t).
<i>imgB</i>	Second frame (time $t+dt$).
<i>pyrA</i>	Buffer for the pyramid for the first frame. If the pointer is not <code>NULL</code> , the buffer must have a sufficient size to store the pyramid from <i>level 1</i> to <i>level #<level></i> (see below); the total size of $(imgSize.width+8)*imgSize.height/3$ bytes will be enough.
<i>pyrB</i>	Similar to <i>pyrA</i> , applies to the second frame. Both parameters comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid and releases the buffer after processing. Otherwise, (the image must be large enough) the function calculates the pyramid and stores it in the buffer unless the <code>CV_LKFLOW_PYR_A[B]_READY</code> flag is set. After the function call both pyramids are calculated and the ready flag for the corresponding image can be set in the next call.
<i>featuresA</i>	Array of points for which the flow needs to be found.
<i>featuresB</i>	Array of 2D points containing calculated new positions of input features in the second image.
<i>count</i>	Number of feature points.
<i>winSize</i>	Size of the search window of each pyramid level.
<i>level</i>	Maximal pyramid level number. If 0, pyramids are not used (single level), if 1, two levels are used, etc.
<i>status</i>	Array. Every element of the array will be set to 1 if the flow for the corresponding feature has been found, 0 otherwise.

<i>error</i>	Array of double numbers containing difference between patches around the original and moved points. Optional parameter, can be NULL.
<i>criteria</i>	Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped.
<i>flags</i>	Miscellaneous flags: <ul style="list-style-type: none">• CV_LKFLOW_PYR_A_READY, pyramid for the first frame is precalculated before the call;• CV_LKFLOW_PYR_B_READY, pyramid for the second frame is precalculated before the call;• CV_LKFLOW_INITIAL_GUESSES, features <i>B</i> array holds initial guesses about new feature locations before the function call.

Discussion

The function calculates the optical flow between two images for the given set of points. The function finds the flow with sup-pixel accuracy.

This chapter describes Estimators functions group.

Overview

Definitions and Motivation

State estimation programs implement a model and an estimator. A model is analogous to a data structure representing relevant information about the visual scene. An estimator is analogous to the software engine that manipulates this data structure to compute beliefs about the world. The OpenCV routines provide two estimators (standard Kalman and condensation).

Models

Many computer vision applications involve repeated estimating i.e., tracking, of the system quantities that change over time. These dynamic quantities are called the system *state*. The system in question can be anything in the scene that happens to be of interest to a particular vision task.

To estimate the state of a system, we assume reasonably accurate knowledge of the system *model* and *parameters*. Parameters are the quantities that describe the model configuration but change at a rate much slower than the state. Parameters are often assumed known and static.

In OpenCV, a state is represented with a vector. In addition to this output of the state estimate routines, there is another vector representing *measurements* that are input to the routines from the sensor data.

For the model, two main parts need to be represented. The first describes the *dynamics* of how we expect the state will change from one time step to the next.

The other thing that needs to be represented is the model of how a measurement vector z_t is obtained from the state.

Estimators

Most estimators have the same general form with repeated propagation and update phases that modify the state's uncertainty as illustrated in Figure 19-1.

Figure 19-1 The ongoing discrete Kalman filter cycle

The time update projects the current state estimate ahead in time. The measurement update adjusts the projected estimate by an actual measurement at that time.

A common, desirable property of an estimator is being unbiased when the probability density of estimate errors has an expected value of 0. There exists an optimal propagation and update formulation that is the best, linear, unbiased, estimator (BLUE) for any given model of the form. This formulation is known as the discrete Kalman estimator, whose standard form is implemented in OpenCV.

Kalman filtering.

The Kalman filter addresses the general problem of trying to estimate the state x of a discrete-time process that is governed by the linear stochastic difference equation

$$x_{k+1} = Ax_k + w_k \quad (1.1)$$

with a measurement z , that is

$$z_k = Hx_k + v_k \quad (1.2)$$

The random variables w_k and v_k respectively represent the process and measurement noise. They are assumed to be independent of each other, white, and with normal probability distributions

$$p(w) = N(0, Q) \quad (1.3)$$

$$p(v) = N(0, R) \quad (1.4)$$

The $N \times N$ matrix A in the difference equation (1.1) relates the state at time step k to the state at step $k+1$, in the absence of process noise. The $M \times N$ matrix H in the measurement equation (1.2) relates the state to the measurement z_k .

We define $x_{\bar{k}}$ (note the "super minus") to be our a priori state estimate at step k given knowledge of the process prior to step k , and x_k to be our a posteriori state estimate at step k given measurement z_k .

We can then define a priori and a posteriori estimate errors as $e_{\bar{k}} = x_k - x_{\bar{k}}$ and $e_k = x_k - x_k$. The a priori estimate error covariance is then $P_{\bar{k}} = E[e_{\bar{k}} e_{\bar{k}}^T]$ and the a posteriori estimate error covariance is $P_k = E[e_k e_k^T]$.

The Kalman filter estimates the process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of noisy measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step. The measurement update equations are responsible for the feedback, i.e., for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate. The time update equations can also be thought of as predictor equations, while the measurement update equations can be thought of as corrector equations. Indeed, the final estimation algorithm resembles that of a predictor-corrector algorithm for solving numerical problems as shown above in Figure 19-1. The specific equations for the time and measurement updates are presented below.

Time update equations

$$x_{\bar{k}+1} = A_k x_k$$

$$P_{\bar{k}+1} = A_k P_k A_k^T + Q_k$$

Measurement update equations:

$$K_k = P_{\bar{k}} H_k^T (H_k P_{\bar{k}} H_k^T + R_k)^{-1}$$

$$x_k = x_{\bar{k}} + K_k (z_k - H_k x_{\bar{k}})$$

$$P_k = (I - K_k H_k) P_{\bar{k}},$$

where K is the so-called Kalman gain matrix, I is the identity operator.

Example 19-1 CvKalman Structure Definition

```

typedef struct CvKalman
{
    int MP;        //Dimension of measurement vector
    int DP;        // Dimension of state vector
    float* PosterState;    // Vector of State of the System in k-th step
    float* PriorState;    // Vector of State of the System in (k-1)-th step
    float* DynamMatr;      // Matrix of the linear Dynamics system
    float* MeasurementMatr;    // Matrix of linear measurement
    float* MNCovariance;    // Matrix of measurement noise covariance
    float* PNCovariance;    // Matrix of process noise covariance
    float* KalmGainMatr;    // Kalman Gain Matrix
    float* PriorErrorCovariance; //Prior Error Covariance matrix
    float* PosterErrorCovariance; //Poster Error Covariance matrix
    float* Temp1;          // Temporary Matrixes
    float* Temp2;
}CvKalman;

```

Reference

cvCreateKalman

*Creates structure for Kalman Filtering and
allocates memory for its members.*

```
void cvCreateKalman(CvKalman** Kalman, int DynamParams, int MeasureParams);
```

Kalman Pointer to the pointer to the structure to be created.

DynamParams Dimension of the state vector.

MeasureParams Dimension of the state vector.

Discussion

The function creates the structure `cvKalman`, allocates memory for its members and stores the pointer to it in `Kalman`.

cvReleaseKalman

Releases the structure `CvKalman` and deallocates memory.

```
void cvReleaseKalman(CvKalman** Kalman);
```

Kalman Pointer to the pointer to the structure to be released.

Discussion

The function releases the structure `cvKalman` and frees the memory previously allocated for the structure.

cvKalmanUpdateByTime

Performs time update for Kalman filtering.

```
void cvKalmanUpdateByTime (CvKalman* Kalman);
```

Kalman Pointer to the pointer to the structure to be released.

Discussion

The function performs updates by the operation time set.

cvKalmanUpdateByMeasurement

Performs measurement update for Kalman filtering.

```
void cvKalmanUpdateByMeasurement (CvKalman* Kalman, CvMat* Measurement);
```

Kalman Pointer to the pointer to the structure to be updated.

Measurement Pointer to the structure CvMat containing the measurement vector.

Discussion

The function performs updates by the operation measurement set.

Condensation algorithm

This section describes the condensation (conditional density propagation) algorithm based on factored sampling. The main idea of the algorithm is using the set of randomly generated samples for probability density approximation. For simplicity we'll describe general principles of Condensation algorithm in case of linear stochastic dynamical system:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{w}_k$$

with a measurement z .

Before the algorithm starts one must generate set of samples \mathbf{x}^n . The samples are randomly generated vectors of states. The function `cvInitSampleSet` does it in `CVLib` implementation.

During the first phase of the condensation algorithm every sample in the set is updated according to the equation (2.1).

Further, when the vector of measurement z obtained, algorithm estimates the conditional probability densities of every sample $P(\mathbf{x}^n|z)$. The `CVLib` implementation of the condensation algorithm enables the user to define various probability density

functions. There is no such special function in the library. After the probabilities are calculated, the user may evaluate, for example, moments of tracked process at the current time step.

Implementation of nonlinear models

If dynamics or measurement of the stochastic system is nonlinear, the user may update the dynamics (A) or measurement (H) matrices, using their Taylor's series on each time step.

Example 19-2 CvConDensation Structure Definition

```
typedef struct
{
    int MP;        //Dimension of measurement vector
    int DP;        // Dimension of state vector
    float* DynamMatr;    // Matrix of the linear Dynamics system
    float* State;    // Vector of State
    int SamplesNum;    // Number of the Samples
    float** flSamples;    // array of the Sample Vectors
    float** flNewSamples;    // temporary array of the Sample Vectors
    float* flConfidence;    // Confidence for each Sample
    float* flCumulative;    // Cumulative confidence
    float* Temp;    // Temporary vector
    float* RandomSample;    // RandomVector to update sample set
    CvRandState* RandS;    // Array of structures to generate random vectors
}CvConDensation;
```

cvCreateConDensation

Creates cvConDensation structure and allocates memory for members.

```
void cvCreateConDensation( CvConDensation* ConDens, int DP, int MP, int
    SamplesNum );
```

<i>ConDens</i>	Pointer to the pointer to the structure to be created.
<i>DynamParams</i>	Dimension of state vector.
<i>MeasureParams</i>	Dimension of the state vector.
<i>SamplesNum</i>	Number of samples.

Discussion

The function creates the structure `cvConDensation`, allocates memory for the structure members, and stores the pointer to it in `ConDens`.

cvReleaseConDensation

Releases CvCondensation structure and deallocates all memory.

```
void cvReleaseConDensation(CvConDensation** ConDens);
```

<i>ConDens</i>	Pointer to the pointer to the structure to be released.
----------------	---

Discussion

The function releases the structure `CvConDensation`, frees all memory previously allocated for the structure.

cvConDensInitSampleSet

Initializes sample set for condensation algorithm.

```
void cvConDensInitSampleSet(CvCondensation* ConDens, CvMat* lowerBound CvMat*  
    upperBound);
```

<i>ConDens</i>	Pointer to a structure to be initialized.
<i>lowerBound</i>	Vector of the lower boundary for each dimension.
<i>upperBound</i>	Vector of the upper boundary for each dimension.

Discussion

The function fills the samples arrays in the structure `CvCondensation` with values within specified ranges.

cvConDensUpdatebyTime

Performs time update for condensation algorithm.

```
void cvConDensUpdateByTime(CvCondensation* ConDens);
```

<i>ConDens</i>	Pointer to the structure to be updated.
----------------	---

Discussion

The function performs update by the operation time set.

This chapter describes functions which together performs POSIT algorithm.

Overview

The POSIT algorithm determines the 6 degree-of-freedom pose of a known, tracked 3D rigid object. Given the projected image coordinates of uniquely-identified points on the object, the algorithm refines an initial pose estimate by iterating with a weak perspective camera model to construct new image points; the algorithm terminates when it reaches a converged image, the pose of which is the solution.

Background

Camera parameters.

Camera parameters are the numbers describing a particular camera configuration. The *intrinsic* camera parameters are those that specify the camera itself; they include the focal length (the distance between the camera lens and the imaging plane), the location of the image center in pixel coordinates, the effective pixel size, and the radial distortion coefficient of the lens. To simplify pose recovery, the focal length is the only intrinsic parameter with which we are concerned, as it is the only one contributing to our model of geometric image formation. The *extrinsic* camera parameters describe the spatial relationship between the camera and the world; they are the rotation matrix and translation vector specifying the transformation between the camera and world reference frames. In the case of pose recovery of a rigid object, the six degree-of-freedom extrinsic parameters are exactly the pose being sought.

Geometric image formation

The link between world points and their corresponding image points is the projection from world space to image space. Due to its generality and usefulness, the most common projection model is the *perspective* (or *pinhole*) model, depicted in Figure 1.

The points in the world are projected onto the image plane according to their distance from the center of projection. Using similar triangles, we can determine that the coordinates of an image point $p_i = (x_i, y_i)$ is related to the coordinates of its world point $P_i = (X_i, Y_i, Z_i)$ as

$$x_i = \frac{f}{Z_i} X_i, y_i = \frac{f}{Z_i} Y_i. \quad (1)$$

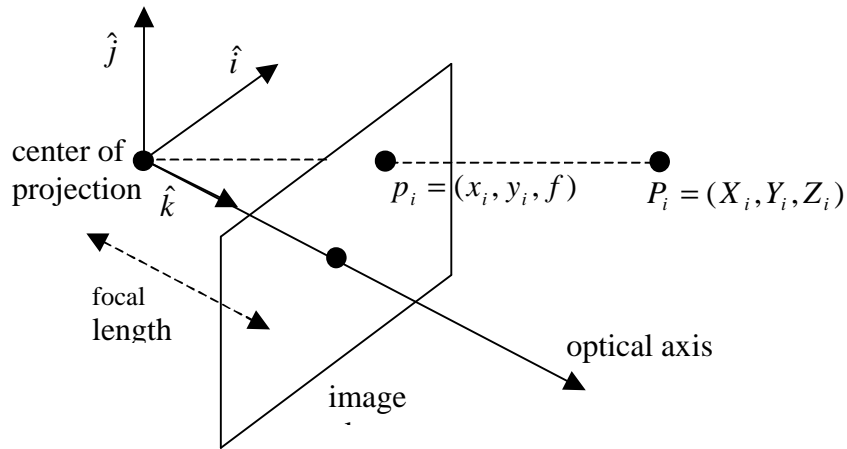


Figure 1. The geometry of perspective projection

The *weak-perspective* projection model simplifies the projection equation by replacing all Z_i with a representative \tilde{Z} so that $s = f/\tilde{Z}$ is a constant scale for all points. The projection equations are then

$$x_i = sX_i, y_i = sY_i. \quad (2)$$

Because this situation can be modeled as an orthographic projection ($x_i = x_i$, $y_i = y_i$) followed by isotropic scaling, weak-perspective projection is sometimes called *scaled orthographic projection*. Weak-perspective is a valid assumption only when the distances between any z_i are much smaller than the distance between the z_i and the center of projection; in other words, the world points are clustered and sufficiently far from the camera. Possible \tilde{z} include any z_i or the average over all z_i .

More detailed explanations of this material can be found in [2].

Pose approximation method

Using weak-perspective projection, we can derive a method for determining approximate pose, termed Pose from Orthography and Scaling (POS) in [1]. First, we choose a reference point P_0 in the world from which we can describe all other world points as vectors: $\vec{P} = P_i - P_0$, as depicted in Figure 2.

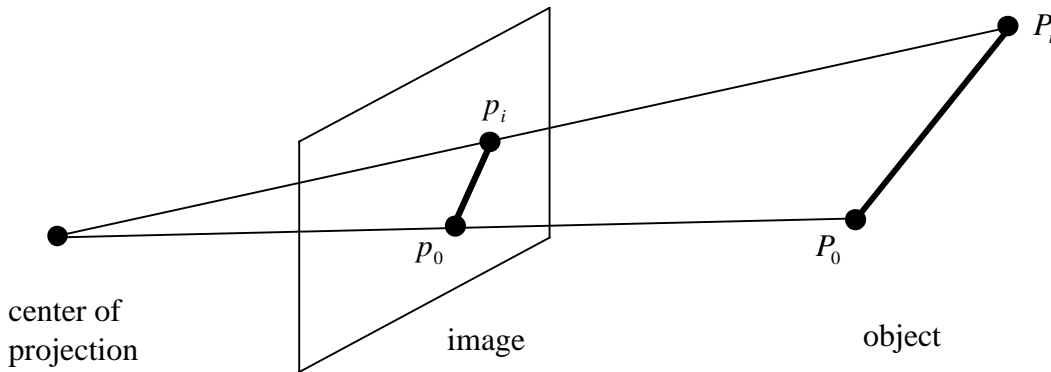


Figure 2. Scaling of vectors in weak-perspective

Similarly, the projection of this point, namely p_0 , is a reference point for the image points: $\vec{p}_i = p_i - p_0$. Using the weak-perspective assumption, the x -component of \vec{p}_i is a scaled-down form of the x -component of \vec{P}_i :

$$x_i - x_0 = s(X_i - X_0) = s(\vec{P}_i \cdot \hat{i}) \quad (3)$$

This is also true for their y -components. If we define I and J as scaled-up versions of the unit vectors \hat{i} and \hat{j} ($I = s\hat{i}$ and $J = s\hat{j}$), we have $x_i - x_0 = \hat{p}_i \cdot I$ and

$$y_i - y_0 = \hat{p}_i \cdot J \quad (4)$$

as two equations for each point for which I and J are unknown. Collecting these equations over all points, we can put them into matrix form as

$$\underline{x} = MI \text{ and } \underline{y} = MJ, \quad (5)$$

where \underline{x} is a vector of \hat{p}_i x -components, \underline{y} is a vector of \hat{p}_i y -components, and M is a matrix whose rows are the \hat{p}_i vectors. These two sets of equations can be further joined to construct a single set of linear equations:

$$[\underline{x} \ \underline{y}] = M[I \ J] \Rightarrow \hat{p}_i C = M[I \ J], \quad (6)$$

where \hat{p}_i is a matrix whose rows are the \hat{p}_i . Now that we have an over-constrained system of linear equations, we can solve for I and J in a least-squares sense as

$$[I \ J] = M^+ \hat{p}_i, \quad (7)$$

where M^+ is the pseudo-inverse of M .

Now that we have I and J , we construct the pose estimate as follows. First, we estimate \hat{i} and \hat{j} as I and J normalized (scaled to unit length). By construction, these are the first two rows of the rotation matrix, and their cross-product is the third row:

$$R = \begin{bmatrix} \hat{i}^T \\ \hat{j}^T \\ (\hat{i} \times \hat{j})^T \end{bmatrix}. \quad (8)$$

The average of the magnitudes of I and J is an estimate of the weak-perspective scale s . From the weak-perspective equations, the world point p_0 in camera coordinates is the image point p_0 in camera coordinates scaled by s :

$$P_0 = p_0/s = [x_0 \ y_0 \ f]/s, \quad (9)$$

which is precisely the translation vector we seek.

Algorithm

The POSIT algorithm is first presented in the paper by DeMenthon and Davis ([1]). In this paper, the authors first describe their POS (Pose from Orthography and Scaling) algorithm. By approximating perspective projection with weak-perspective projection POS produces a pose estimate from a given image. POS can be repeatedly used by constructing a new weak perspective image from each pose estimate and feeding it into the next iteration. The calculated images are estimates of the initial perspective image with successively smaller amounts of “perspective distortion” so that the final image contains no such distortion. They term this iterative use of POS as POSIT (POS with Iterations).

POSIT requires three pieces of known information. First, the *object model* consists of N points, each with unique 3-D coordinates. N must be greater than 3, and the points must be nondegenerate (noncoplanar) to avoid algorithmic difficulties. (Better results are achieved by using more points and by choosing points as far from coplanarity as possible.) The object model is an $N \times 3$ matrix. Second, the *object image* is the set of 2D points resulting from a camera projection of the model points onto an image plane; it is a function of the object’s current pose. The object image is an $N \times 2$ matrix. Finally, the focal length of the camera must be known.

Given the object model and the object image, the algorithm proceeds as follows. First, the object image is assumed to be a weak perspective image of the object, from which a least-squares pose approximation is calculated via the object model pseudoinverse. From this approximate pose the object model is projected onto the image plane to construct a new weak perspective image. From this image a new approximate pose is found using least-squares, which in turn determines another weak perspective image, etc. For well-behaved inputs, this procedure converges to an unchanging weak perspective image, whose corresponding pose is the final calculated object pose.

```
POSIT (imagePoints, objectPoints, focalLength) {  
    count = converged = 0;  
    modelVectors = modelPoints - modelPoints(0);  
    oldWeakImagePoints = imagePoints;  
    while (!converged) {  
        if (count == 0)  
            imageVectors = imagePoints - imagePoints(0);
```

```

else {
    weakImagePoints = imagePoints .*
        ((1 + modelVectors*row3/translation(3)) * [1
1]);

    imageDifference = sum(sum(abs( round(weakImagePoints) -
        round(oldWeakImagePoints))));

    oldWeakImagePoints = weakImagePoints;
    imageVectors = weakImagePoints - weakImagePoints(0);
}
[I J] = pseudoinverse(modelVectors) * imageVectors;
row1 = I / norm(I);
row2 = J / norm(J);
row3 = crossproduct(row1, row2);
rotation = [row1; row2; row3];
scale = (norm(I) + norm(J)) / 2;
translation = [imagePoints(1,1); imagePoints(1,2); focalLength]
/
    scale;
converged = (count > 0) && (diff < 1);
count = count + 1;
}
return {rotation, translation};
}

```

Because the first step assumes the object image is a weak perspective image of the object, a valid assumption only for an object sufficiently far from the camera so that “perspective distortions” are insignificant, for such objects the correct pose is recovered immediately and convergence occurs at the second iteration. For less ideal situations, the pose is quickly recovered after several iterations. However, convergence is not guaranteed when perspective distortions are significant, such as when an object is close to the camera with pronounced foreshortening. DeMenthon and Davis state that “convergence seems to be guaranteed if the image features are at a distance from the image center shorter than the focal length.” Fortunately, this occurs for most realistic camera and object configurations.

Reference

cvCreatePOSITObject

Initializes structure containing object information.

```
void cvCreatePOSITObject( CvPoint3D32f* points, int numPoints, CvPOSITObject**  
    ppObject );
```

<i>points</i>	Pointer to the points of the 3D object model.
<i>numPoints</i>	Number of object points.
<i>ppObject</i>	Address of the pointer to object structure. The memory for the structure will be allocated by this function. Use the function <code>cvReleasePOSITObject</code> to release the memory.

Discussion

The function is intended to allocate memory for the object structure and compute the object inverse matrix.

These data will be stored in the structure `CvPOSITObject`, internal for OpenCV, which means that the user may not directly manage the structure. The user must only create a pointer to this structure and pass it to the function.

Object is defined as a set of points given in a coordinate system. The function `cvPOSIT` computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function `cvReleasePOSITObject` must be called to free memory.

cvPOSIT

Implements POSIT algorithm.

```
void cvPOSIT( CvPoint2D32f* imagePoints, CvPOSITObject* pObject, double
              focalLength, CvTermCriteria criteria, CvMatrix3_3* rotation, CvPoint3D32f*
              translation );
```

<i>imagePoints</i>	Pointer to the object points projections on the 2D image plane.
<i>pObject</i>	Pointer to the object structure.
<i>focalLength</i>	Focal length of the camera used.
<i>criteria</i>	Termination criteria of the iterative POSIT algorithm.
<i>rotation</i>	Matrix of rotations.
<i>translation</i>	Translation vector.

Discussion

Image coordinates are given in camera-related coordinate system. The focal length of camera must be defined by camera calibration functions. At every iteration of the algorithm new perspective projection of estimated pose is computed.

Difference norm between two projections is the maximal distance between correspondent points. The parameter *criteria.epsilon* serves to stop the algorithm if the difference is small.

cvReleasePOSITObject

Releases free memory employed by object structure.

```
void cvReleasePOSITObject( CvPOSITObject** ppObject );
```

<i>ppObject</i>	Address of pointer to object structure.
-----------------	---

Discussion

The function is used to release memory previously allocated by the function `cvCreatePOSITObject`.

This chapter describes functions that operate on multi-dimensional histograms.

Overview

The Computer Vision Library functions operate on a single format for histograms in memory. This format consists of a header of type `CvHistogram` containing the information for all histogram attributes. The C language definition for the `CvHistogram` structure is given below.

Example 21-1 `CvHistogram` Structure Definition

```
typedef struct CvHistogram
{
    int      header_size; /* header's size          */
    CvHistType type;      /* type of histogram    */
    int      flags;       /* histogram's flags     */
    int      c_dims;      /* histogram's dimension */
    int      dims[CV_HIST_MAX_DIM];
                        /* every dimension size */
    int      mdims[CV_HIST_MAX_DIM];
                        /* coefficients for fast
                        access to element      */
                        /* &m[a,b,c] = m + a*mdims[0] +
                        b*mdims[1] + c*mdims[2] */
    float*   thresh[CV_HIST_MAX_DIM];
                        /* thresholds for every
```

```
                                dimension */
float*  array; /* all the histogram data, expanded into
               the single row */
struct  CvNode* root; /* tree - histogram data */
CvSet*   set; /* pointer to memory storage
              (for tree data) */
int*  chdims[CV_HIST_MAX_DIM];
                                /* cache data for fast calculating */
} CvHistogram;
```

Reference

cvCreateHist

Creates histogram.

```
void cvCreateHist( int c_dims, int* dims, CvHistType type, CvHistogram** hist
);
```

<i>c_dims</i>	Histogram dimension number.
<i>dims</i>	Dimension size array.
<i>type</i>	Histogram type (must be CV_HIST_ARRAY or CV_HIST_TREE).
<i>hist</i>	Pointer to the histogram to be created.

Discussion

The function creates a histogram of the specified size and returns (via output parameter) pointer to created histogram.

cvReleaseHist

Releases histogram header and underlying data.

```
void cvReleaseHist, ( CvHistogram** hist );
```

hist Pointer to the released histogram.

Discussion

The function releases the histogram header and underlying data.

cvMakeHistHeaderForArray

Initializes histogram header.

```
void cvMakeHistHeaderForArray( int c_dims, int* dims, CvHistogram* hist,
float* data );
```

c_dims Histogram dimension number.

dims Dimension size array.

hist Pointer to the histogram to be created.

data Pointer to the source data histogram.

Discussion

Initializes the histogram header and sets the data pointer to the given value (*data*). The histogram must have the type CV_HIST_ARRAY.

cvQueryHistValue_1D

Queries value of histogram bin.

```
float cvQueryHistValue_1D( CvHistogram* hist, int idx0 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Address of a required element.
<i>idx1</i>	Address of a required element.
<i>idx2</i>	Address of a required element.
<i>idx</i>	Address array of a required element.

Discussion

The function searches the value of the histogram bin.

Return values

The function returns the value of the specified histogram bin. If the histogram doesn't store null bins and the bin is not present in the histogram, the function returns 0.

cvQueryHistValue_2D

Queries value of histogram bin.

```
float cvQueryHistValue_2D( CvHistogram* hist, int idx0, int idx1 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Address of a required element.
<i>idx1</i>	Address of a required element.
<i>idx2</i>	Address of a required element.
<i>idx</i>	Address array of a required element.

Discussion

The function searches the value of the histogram bin.

Return values

The function returns the value of the specified histogram bin. If the histogram doesn't store null bins and the bin is not present in the histogram, the function returns 0.

cvQueryHistValue_3D

Queries value of histogram bin.

```
float cvQueryHistValue_3D( CvHistogram* hist, int idx0, int idx1, int idx2 );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Address of a required element.
<i>idx1</i>	Address of a required element.
<i>idx2</i>	Address of a required element.
<i>idx</i>	Address array of a required element.

Discussion

The function searches the value of the histogram bin.

Return values

The function returns the value of the specified histogram bin. If the histogram doesn't store null bins and the bin is not present in the histogram, the function returns 0.

cvQueryHistValue_nD

Queries value of histogram bin.

```
float cvQueryHistValue_nD( CvHistogram* hist, int* idx );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Address of a required element.
<i>idx1</i>	Address of a required element.
<i>idx2</i>	Address of a required element.
<i>idx</i>	Address array of a required element.

Discussion

The function searches the value of the histogram bin.

Return values

The function returns the value of the specified histogram bin. If the histogram doesn't store null bins and the bin is not present in the histogram, the function returns 0.

cvGetHistValue_1D, cvGetHistValue_2D, cvGetHistValue_3D, cvGetHistValue_nD

Returns pointer to histogram bin.

```
float* cvGetHistValue_1D( CvHistogram* hist, int idx0 );
float* cvGetHistValue_2D( CvHistogram* hist, int idx0, int idx1 );
float* cvGetHistValue_3D( CvHistogram* hist, int idx0, int idx1, int idx2 );
float* cvGetHistValue_nD( CvHistogram* hist, int* idx );
```

<i>hist</i>	Pointer to the source histogram.
<i>idx0</i>	Address of a required element.

<i>idx1</i>	Address of a required element.
<i>idx2</i>	Address of a required element.
<i>idx</i>	Address array of a required element.

Discussion

The function returns the pointer to the histogram bin, given its coordinates. If the bin is not present, it is created and initialized with 0.

Return values

The pointer `NULL` is returned when arguments are invalid or there is no memory available.

cvGetMinMaxHistValue

Finds minimum and maximum histogram bins.

```
void cvGetMinMaxHistValue( CvHistogram* hist, int* idx_min, float* value_min,
                           int* idx_max, float* value_max );
```

<i>hist</i>	Pointer to the histogram.
<i>idx_min</i>	Pointer to the array of coordinates for minimum. If not <code>NULL</code> , must have <i>hist</i> -> <i>c_dims</i> elements.
<i>value_min</i>	Pointer to the minimum value of the histogram. Can be <code>NULL</code> .
<i>idx_max</i>	Pointer to the array of coordinates for maximum. If not <code>NULL</code> , must have <i>hist</i> -> <i>c_dims</i> elements.
<i>value_max</i>	Pointer to the maximum value of the histogram. Can be <code>NULL</code> .

Discussion

The function finds the minimum and maximum histogram bins and their positions.

cvNormalizeHist

Normalizes histogram.

```
void cvNormalizeHist( CvHistogram* hist, float factor );
```

hist Pointer to the histogram.

factor Normalize factor.

Discussion

The function normalizes the histogram, such that the sum of histogram bins becomes equal to *factor*.

cvThreshHist

Thresholds histogram.

```
void cvThreshHist( CvHistogram* hist, float thresh );
```

hist Pointer to the histogram.

thresh Threshold value.

Discussion

The function clears histogram bins that are below the specified level.

cvCompareHist

Compares two histograms.

```
double cvCompareHist( CvHistogram* hist1, CvHistogram* hist2, CvCompareMethod  
                      method );
```

hist1 First histogram.

<i>hist2</i>	Second histogram.
<i>method</i>	Comparison method, may be any of the ones listed below: <ul style="list-style-type: none"> • CV_COMP_CORREL; • CV_COMP_CHISQR; • CV_COMP_INTERSECT.

Discussion

The function compares two histograms using specified method.

$$\text{CV_COMP_CORREL } result = \frac{\sum_i \hat{q}_i \hat{v}_i}{\sqrt{\sum_i \hat{q}_i^2 * \sum_i \hat{v}_i^2}}$$

$$\text{CV_COMP_CHISQR } result = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i}$$

$$\text{CV_COMP_INTERSECT } result = \sum_i \min(q_i, v_i)$$

Return values

The function returns a value that characterizes similarity or difference of two histograms.

cvCopyHist

Copies histogram.

```
void cvCopyHist( CvHistogram* src, CvHistogram** dst );
```

<i>src</i>	Source histogram.
<i>dst</i>	Pointer to destination histogram.

Discussion

Makes a copy of the histogram. If the second histogram pointer **dst* is null, it will be allocated and the pointer will be stored at **dst*. Otherwise both histograms must have equal types and sizes.

cvSetHistThresh

Sets bounds of histogram bins.

```
void cvSetHistThresh( CvHistogram* hist, float** thresh, int uniform );
```

<i>hist</i>	Destination histogram.
<i>thresh</i>	Pointer to the array of threshold values.
<i>uniform</i>	Uniform flag.

Discussion

The function sets bounds of histogram bins. If the parameter *uniform* is not equal to 0, *thresh[i][0]* is the minimum boundary and *thresh[i][1]* is the maximum boundary of the *i*th histogram dimension. The input value in the *i*th plane is considered then. The histogram or back project are calculated if within the specified boundaries. Number of bins per each dimension is specified when histogram is created.

cvCalcHist, cvCalcHistMask

Calculates histogram.

```
void cvCalcHist( IplImage** img, CvHistogram* hist, int dont_clear );
void cvCalcHistMask( IplImage** img, IplImage* mask, CvHistogram* hist, int
    dont_clear );
```

<i>img</i>	Source images.
<i>hist</i>	Pointer to the histogram.

mask Mask image (for mask functions).
dont_clear Clear flag.

Discussion

The function calculates the histogram of the array of single-channel images. If the parameter *dont_clear* parameter is 0, then the histogram is cleared before calculation; otherwise the histogram is simply updated. For flavors of the function that support mask, only pixels with a non-zero value in the mask at the corresponding position are considered.

cvCalcBackProject

Calculates back project.

```
void cvCalcBackProject( IplImage** img, IplImage* dst_img, CvHistogram* hist);
```

img Source images array.
dst_img Destination image.
hist Source histogram.

Discussion

The function calculates the back project of the histogram. For each group of pixels taken from the same position from all input single-channel images, functions put to the destination image value of the histogram bin, coordinates of which are determined by the values of pixels in this input group. From the statistical point of view, output image pixel values characterize the probability of input pixels being at the same position.

cvCalcBackProjectPatch

Calculating back project patch of histogram.

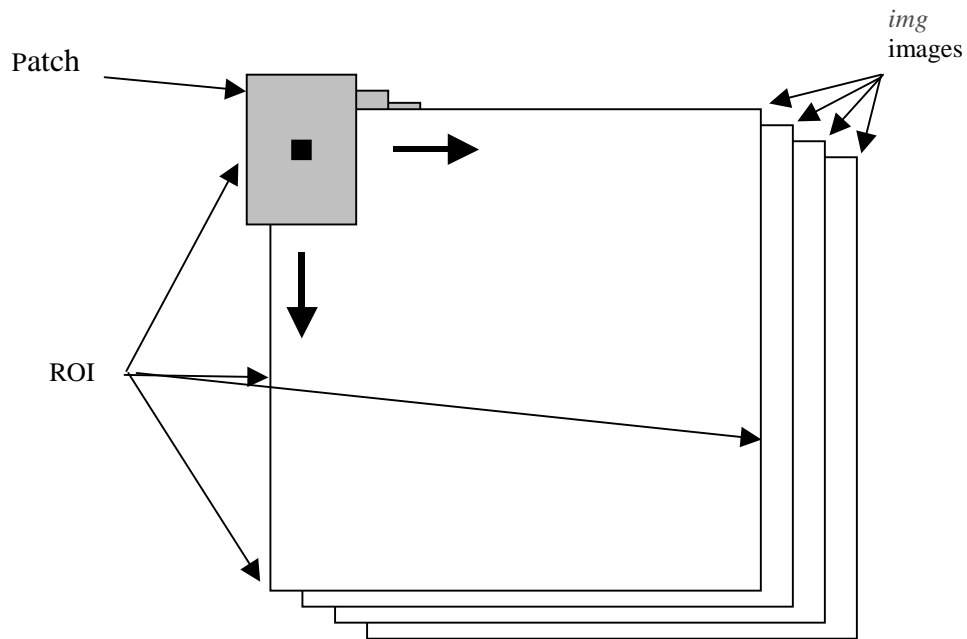
```
void cvCalcBackProjectPatch( IplImage** img, IplImage* dst, CvSize range,  
    CvHistogram* hist, CvCompareMethod method, float norm_factor );
```

<i>img</i>	Source images array.
<i>dst</i>	Destination image.
<i>range</i>	Range size (see below).
<i>hist</i>	Probabilistic model.
<i>method</i>	Comparing method.
<i>norm_factor</i>	Normalize factor.

Discussion

A *img* array is created by taking measures from an image *I* at each location over a ROI. These measures might be one or more of hue, *x* derivative, *y* derivative, Laplacian filter, oriented Gabor filter etc. Each measure output is collected into its own separate image. The *img* image array is a collection of these measure images. A multi-dimensional histogram *hist* is constructed by sampling from the *img* image array, the final histogram is normalized. The *hist* histogram will have as many dimensions as image planes in the image *measuresI*.

Each new image is taken measures of and then converted into an *img* image array over a chosen ROI. Histograms are taken from this *img* image in an area covered by a “patch” with anchor at center as shown in Figure 3-1, the histogram is normalized (uses the parameter *norm_factor*) so that it may be compared with *hist*. The calculated histogram is compared to the model histogram; *hist* uses the function *cvCompareHist* (the parameter *method*). The resulting output is placed at the location corresponding to the patch anchor in the probability image *dst*. This process is repeated as the patch is slid over the ROI. Note that many calculations can be saved by subtracting trailing pixels covered by the patch and adding in newly covered pixels.

Figure 21-1 Array of images *img*

Each image of the image array *img* shown on the figure above stores the corresponding element of a multi-dimensional measurement vector is shown. Histogram measures are drawn from measure vectors over a patch with anchor at center. A multi-dimensional histogram *hist* is used via the function `cvCompareHist` to calculate the output at the patch anchor. The patch is slid around until the values are calculated over the whole ROI.

cvCalcEMD

Computes Earth mover distance.

```
void cvCalcEMD(float* signature1, int size1, float* signature2, int size2, int
    dims, CvDisType dist_type, float (*dist_func)( float* f1, float* f2, void*
    user_param), float* emd, float* lower_bound, void* user_param);
```

signature1 First signature, array of $size1 * (dims + 1)$ elements.

signature2 Second signature, array of $size2 * (dims + 1)$ elements.

dims Number of dimensions in feature space. If 0, then *signature1* and *signature2* are considered simple 1D histograms. Otherwise both signatures must look as follows:

```
(weight_i0, x0_i0, x1_i0, ..., x(dims-1)_i0,
weight_i1, x0_i1, x1_i1, ..., x(dims-1)_i1,
...
weight_(size1-1), x0_(size1-1), x1_(size1-1), ...,
x(dims-1)_(size1-1)),
```

where *weight_ik* is the weight of *ik* cluster. *x0_ik*, ..., *x(dims-1)_ik* are coordinates of the cluster *ik*.

dist_type CV_DIST_L1, CV_DIST_L2, CV_DIST_C stand for one of the standard metrics. ((CvDisType)-1) stands for a user-defined distance function, which passes two coordinate vectors and the user parameter and returns the distance between these feature points.

emd Pointer to the calculated *emd* distance.

lower_bound Pointer to the calculated lower boundary. If 0, only *emd* is calculated. Otherwise, if the calculated lower boundary is greater than or equal to the value stored at this pointer, then the true *emd* is not calculated, but is set to that *lower_bound*.

Discussion

Histograms and Signatures

Histograms represent a simple statistical description of an object, e.g., an image. The object characteristics are measured during iterating through that object: for example, color histograms for the image are built from pixel values in one of the color spaces. We quantize all the possible values of that multi-dimensional characteristic on each coordinate. If the quantized characteristic may take k_1 different values on the first coordinate, k_2 values on second ... and k_n on the last one, the resulting histogram will

have the size $size = \prod_{i=1}^n k_i$

The histogram is stored in an array and the array element, otherwise called a histogram bin, $[i_1, i_2 \dots i_n]$ contains a number of measurements done for the object, which quantized characteristics value is i_1 on first coordinate, i_2 on second etc. We can compare objects using their histograms:

$$D_{L_1}(H, K) = \sum_i |h_i - k_i|, \text{ or}$$

$$D(H, K) = \sqrt{(\bar{h} - \bar{k})^T A (\bar{h} - \bar{k})}.$$

But these methods suffer from several disadvantages. D_{L_1} sometimes gives too small difference when there is no exact correspondence between histogram bins, i.e., if the bins of one histogram are slightly shifted. On the other hand, D_{L_2} gives too large difference due to cumulative property.

Another drawback of pure histograms is large space required, especially for higher-dimensional characteristics. The solution is to store not all histogram bins, but only the ones that are non-zero, or just the ones with the highest score. Generalization of histograms is termed *signature* and defined in the following way:

1. Characteristic values with rather fine quantization are gathered;
2. Only non-zero “bins” are dynamically stored.

This can be implemented using hash-tables, balanced trees, or other “sparse” structures. After processing a set of “clusters” is available, each of them is characterized by the coordinates and weight, i.e., number of measurements in the

neighborhood. We can further reduce the signature size by removing clusters with small weight. Although these structures can't be compared using formulas written above, there exists a robust comparison method described in [1] called Earth Mover Distance.

Earth Mover Distance (EMD) definition

Physically, two signatures can be viewed as two systems - earth masses, spread by several localized pieces. Each piece (cluster) has some coordinates in space and weight, the earth mass it contains. The distance between two systems can be measured then, as a minimal work, needed to get the second configuration from the first or vice versa. To get metric, invariant to scale, we can divide the result by the total mass of the system.

Mathematically, it can be formulated as following. Consider m suppliers and n consumers.

Let the capacity of i^{th} supplier be x_i and the capacity of j^{th} consumer be y_j . Also, let the ground distance between i^{th} supplier and j^{th} consumer be c_{ij} . The following restrictions must be satisfied:

$$x_i \geq 0, y_j \geq 0, c_{i,j} \geq 0,$$

$$\sum x_i \geq \sum y_j,$$

$$0 \leq i < m, 0 \leq j < n.$$

Then the task is to find the flow matrix $\|f_{ij}\|$, where f_{ij} is the amount of "earth", transferred from i^{th} supplier to j^{th} consumer. This flow must satisfy the restrictions bellow:

$$f_{i,j} \geq 0,$$

$$\sum_i f_{i,j} \leq x_i,$$

$$\sum_j f_{i,j} = y_j$$

and minimize the overall cost:

$$\min \sum_i \sum_j c_{i,j} f_{i,j}$$

If $\|f_{ij}^*\|$ is the optimal flow, then Earth Mover Distance is defined as

$$EMD(x, y) = \frac{\sum_i \sum_j c_{i,j} f_{i,j}}{\sum_i \sum_j f_{i,j}}$$

Note, that the task on finding the optimal flow is actually well known transportation problem, which can be solved, for example, using the simplex method.

Example Ground Distances

As we have seen in the section above, physically intuitive distance between two systems could be found if we can measure the distance between their elements. The last distance is called ground distance and if it is true metric then the resultant distance between systems is a metric too. The choice of the ground distance depends on the concrete task as well as the choice of the coordinate system for the measured characteristic. In [2], [3] three different distances are considered. The first is used for human-like color discrimination between pictures. CIE Lab model represents colors in a way when a simple Euclidean distance gives true human-like discrimination between colors. So, if we convert image pixels into CIE Lab format, i.e., represent colors as 3D-vectors (L,a,b), and quantize them (in 25 segments on each coordinate in [2]), we'll produce a color-based signature of the image. Although in experiment, made in [2], the maximal number of non-zero bins could be $25 \times 25 \times 25 = 15625$, the average number of clusters was ~ 8.8 , that is, resulting signatures were very compact! The second example is more complex. Not only the color values are considered, but also the coordinates of the corresponding pixels, which makes it possible tell pictures containing similar color palette but representing different color regions placements from one another: e.g., green grass at the bottom and blue sky on top vs. green forest on top and blue lake at the bottom. 5D space is used and metric is:

$[(\Delta L)^2 + (\Delta a)^2 + (\Delta b)^2 + \lambda((\Delta x)^2 + (\Delta y)^2)]^{1/2}$, where λ regulates importance of the spatial correspondence. When $\lambda = 0$, we get the first metric.

The third example is related to texture metrics.

In the example Gabor transform is used to get the 2D-vector texture descriptor (l, m) , which is a log-polar characteristic of the texture. Then, no-invariance ground distance is defined as (α, L, M) , the scale parameter of Gabor transform, L , the number of different angles used (angle resolution), M , the number of scales used (scale resolution):

$$d((l_1, m_1), (l_2, m_2)) = |\Delta l| + \alpha |\Delta m|, \Delta l = \min(|l_1 - l_2|, L - |l_1 - l_2|), \Delta m = |m_1 - m_2|$$

To get invariance to scale and rotation, the user may calculate minimal EMD for several scales, rotations:

$$(l_1, m_1), (l_2, m_2)$$

$$EMD(t_1, t_2) = \min_{\substack{0 \leq l_0 < L \\ -M < m_0 < M}} EMD(t_1, t_2, l_0, m_0)$$

Where d is measured as in previous case, but Δl and Δm look slightly different:

$$\Delta l = \min(|l_1 - l_2 + l_0(\text{mod } L)|, L - |l_1 - l_2 + l_0(\text{mod } L)|), \Delta m = |m_1 - m_2 + m_0|$$

Lower boundary for EMD

If ground distance is metric and distance between points can be calculated via the norm of their difference, and total suppliers' capacity is equal to total consumer' capacity, then it is easy to calculate lower bound of EMD. It is because:

$$\begin{aligned} \sum_i \sum_j c_{i,j} f_{i,j} &= \sum_i \sum_j \|p_i - q_j\| f_{i,j} = \sum_i \sum_j \|p_i - q_j\| f_{i,j} \\ &\geq \left\| \sum_i \sum_j \|p_i - q_j\| f_{i,j} \right\| = \left\| \sum_i \left(\sum_j f_{i,j} \right) p_i - \sum_j \left(\sum_i f_{i,j} \right) q_j \right\| \\ &= \left\| \sum_i x_i p_i - \sum_j y_j q_j \right\| \end{aligned}$$

As it can be seen, the last expression is the distance between the mass centers of the systems.

Thus, it is possible, using this lower boundary for EMD distance, to eject poor candidates, when searching in the large image database, rather fast.

Gesture Recognition

22

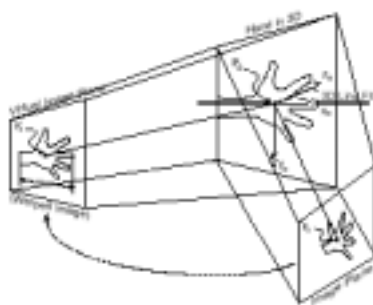
This chapter describes specific functions for the static gesture recognition technology.

Overview

The gesture recognition algorithm can be divided into four main components as illustrated in Fig. 22-1.

The first component computes the 3D arm pose from range image data that may be obtained from the standard stereo correspondence algorithm. The process includes 3D-line fitting, finding the arm position along the line and creating the arm mask image.

Figure 22-1 Gesture Recognition Algorithm



The second component produces a frontal view of the arm image and arm mask through a planar homograph transformation. The process consists of the homograph matrix calculation and warping image and image mask (Fig.22-2).

The third component segments the arm from the background based on the probability density estimate that a pixel with a given hue and saturation value belongs to the arm. For this 2D-image histogram, image mask histogram and probability density histogram are calculated. After this initial estimate is iteratively refined using the maximum likelihood approach and morphology operations (Fig.22-3).

Figure 22-2 Arm location and image warping

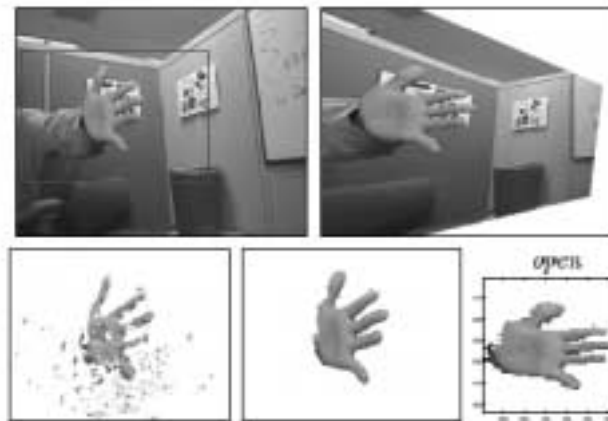
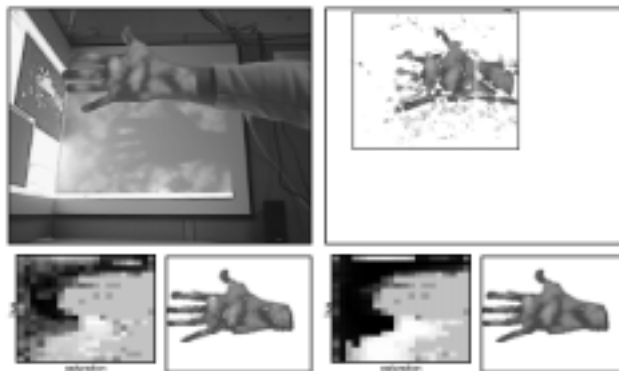


Figure 22-3 Arm segmentation by probability density estimation

The fourth step is the recognition step when seven area normalized image moments Huare calculated using the resulting image mask. These invariants are used to match masks by the Mahalanobis distance metric calculation.

The functions operate with specific data of several types. Range image data are a set of 3D points in the world coordinate system calculated via the stereo correspondence algorithm. The second data type is a set of the original image indices of this set of 3D points, i.e., projections on the image plane. The functions of this group enable the user to locate the arm region in a set of 3D points (the functions `cvFindHandRegion`, `cvFindHandRegionA`), create image mask from a subset of 3D points and associated subset indexes around the arm center (the function `cvCreateHandMask`), calculate the homography matrix for the initial image transformation from the image plane to the plane defined by the frontal arm plane (the function `cvCalcImageHomography`), and calculate the probability density histogram for the arm location)the function `cvCalcProbDensity`).

Reference

cvFindHandRegion

Finds arm region in 3D range image data.

```
void cvFindHandRegion( CvPoint3D32f* points, int count, CvSeq* indexs, float*  
    line, CvSize2D32f size, int flag, CvPoint3D32f* center, CvMemStorage*  
    storage, CvSeq** numbers);
```

<i>points</i>	Pointer to the input 3D point data.
<i>count</i>	Numbers of the input points.
<i>indexs</i>	Sequence of the input points indices on the initial image.
<i>line</i>	Pointer to the input points approximation line.
<i>size</i>	Size of the initial image.
<i>flag</i>	Flag of the arm orientation.
<i>center</i>	Pointer to the output arm center.
<i>storage</i>	Pointer to the memory storage.
<i>numbers</i>	Pointer to the output sequence of the points indices.

Discussion

This function finds the arm region in 3D range image data. The coordinates of the points must be defined in the world coordinates system. Each input point has user-available transform indices on the initial image (*indexs*). The function finds the arm region along the approximation line line from the left, if *flag* = 0, or from right, if *flag* = 1, in the points maximum accumulation by the points projection histogram calculation. Also the function calculates the center of the arm region and the indices of the points that lie near the arm center. The function assumes that the arm length is equal to about 0.25m in the world coordinate system.

cvFindHandRegionA

Finds arm region in 3D range image data and defines arm orientation.

```
void cvFindHandRegionA( CvPoint3D32f* points, int count, CvSeq* indexs, float*
    line, CvSize2D32f size, int j_center, CvPoint3D32f* center, CvMemStorage*
    storage, CvSeq** numbers);
```

<i>points</i>	Pointer to the input 3D point data.
<i>count</i>	Numbers of the input points.
<i>indexs</i>	Sequence of the input points indices on the initial image.
<i>line</i>	Pointer to the input points approximation line.
<i>size</i>	Size of the initial image.
<i>j_center</i>	Input <i>j</i> -index of the initial image center.
<i>center</i>	Pointer to the output arm center.
<i>storage</i>	Pointer to the memory storage.
<i>numbers</i>	Pointer to the output sequence of the points indices.

Discussion

This function finds the arm region in the 3D range image data and defines the arm orientation (left or right). The coordinates of the points must be defined in the world coordinates system. The input parameter *j_center* is the index *j* of the initial image center in pixels (*width*/2). Each input point has user-available transform indices on the initial image (*indexs*). The function finds the arm region along approximation line from the left or from the right in the points maximum accumulation by the points projection histogram calculation. Also the function calculates the center of the arm region and the indices of points that lie near the arm center. The function assumes that the arm length is equal to about 0.25m in the world coordinate system.

cvCreateHandMask

Creates arm mask on image plane.

```
void cvCreateHandMask(CvSeq* numbers, IplImage *img_mask, CvRect *roi);
```

numbers Sequence of the input points indices on the initial image.

img_mask Pointer to the output image mask.

roi Pointer to the output arm ROI.

Discussion

This function creates the arm mask on the image plane. The pixels of the result mask associated with the set of indices on the initial image (*indexes*) will have the maximum unsigned char value (255). All remaining pixels will have the minimum unsigned char value (0). The output image mask (*img_mask*) has to have the `IPL_DEPTH_8U` type and the numbers of channels is 1.

cvCalcImageHomography

Calculates homography matrix.

```
void cvCalcImageHomography(float *line, CvPoint3D32f* center, float  
    intrinsic[3][3], float homography[3][3]);
```

line Pointer to the input 3D line.

center Pointer to the input arm center.

intrinsic Matrix of the intrinsic camera parameters.

homography Output homography matrix.

Discussion

This function calculates the homograph matrix for the initial image transformation from image plane to the plane, defined by 3D arm line (fig.9.1). If $n_1 = (n_x, n_y)$ and $n_2 = (n_x, n_z)$ are coordinates of the normals of the 3D line projection of plane xy and xz , then the result image homograph matrix is calculated as

$H = A \cdot (R_h + (I_{3 \times 3} - R_h) \cdot \bar{x}_h \cdot [0, 0, 1]) \cdot A^{-1}$, where R_h is the 3x3 matrix $R_h = R_1 \cdot R_2$, and $R_1 = [n_1 \times u_z, n_1, u_z]$, $R_2 = [u_y \times n_2, u_y, n_2]$, $u_z = [0, 0, 1]^T$, $u_y = [0, 1, 0]^T$, $\bar{x}_h = \frac{T_h}{T_z} = \left[\frac{T_x}{T_z}, \frac{T_y}{T_z}, 1 \right]^T$, where (T_x, T_y, T_z) is the arm center coordinates in the world coordinate system. A is the intrinsic camera parameters matrix

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

The diagonal entries f_x and f_y are the camera focal length in units of horizontal and vertical pixels and the two remaining entries c_x, c_y are the principal point image coordinates.

cvCalcProbDensity

Calculates arm mask probability density on image plane.

```
void cvCalcProbDensity (CvHistogram* hist, CvHistogram* hist_mask,
    CvHistogram* hist_dens );
```

<i>hist</i>	Input image histogram.
<i>hist_mask</i>	Input image mask histogram.
<i>hist_dens</i>	Result probability density histogram.

Discussion

This function calculates the arm mask probability density from the two 2D-histograms. The input histograms have to be calculated in the two channels on the initial image. If $\{h_{ij}\}$ and $\{hm_{ij}\}$, $1 \leq i \leq B_i$, $1 \leq j \leq B_j$ input histogram and mask histogram, when the result probability density histogram p_{ij} is calculated as

$$p_{ij} = \begin{cases} \frac{m_{ij}}{h_{ij}} \cdot 255, & \text{if } h_{ij} \neq 0, \\ 0, & \text{if } h_{ij} = 0, \\ 255, & \text{if } m_{ij} > h_{ij} \end{cases}$$

So the values of the p_{ij} are between 0 and 255.

cvMaxRect

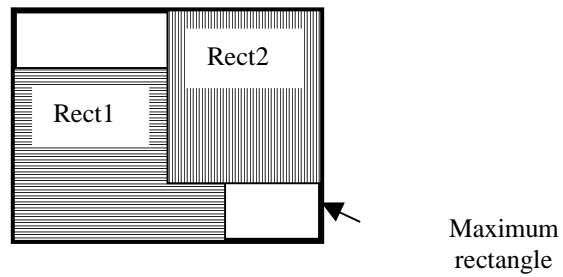
Calculates the maximum rectangle.

```
void cvMaxRect (CvRect* rect1, CvRect* rect2, CvRect* max_rect );
```

rect1 First input rectangle.
rect2 Second input rectangle.
max_rect Result maximum rectangle.

Discussion

This function calculates the maximum rectangle for two input rectangles (Fig. 22-4).

Figure 22-4 Maximum rectangular for two input rectangles

This chapter describes functions for matrix operations.

Overview

Example 23-1 CvMat Structure Definition

```
typedef struct CvMat
{
    int rows;           // number of rows
    int cols;           // number of cols
    CvMatType type;     // type of matrix
    int step;           // not used
    union
    {
        float* fl; //pointer to the float data
        double* db; //pointer to the double

    }data;
}CvMat
```

Example 23-2 CvMatArray Structure Definition

```
typedef struct CvMatArray
{
    int rows; //number of rows
```

```
int cols; //number pf cols
int type; // type of matrices
int step; // not used
int count; // number of matrices in aary
union
{
    float* fl;
    float* db;
}data; // pointer to matrix array data
}CvMatArray
```

Reference

cvmAlloc

Allocates memory for matrix data.

```
void cvmAlloc (CvMat*  mat);
```

mat Pointer to the matrix for which memory must be allocated.

Discussion

The function allocates memory for matrix data.

cvmAllocArray

Allocates memory for matrix array data.

```
void cvmAllocArray (CvMatArray*  matAr);
```

matAr Pointer to the matrix array for which memory must be allocated.

Discussion

The function allocates memory for matrix array data.

cvmFree

Frees memory allocated for matrix data.

```
void cvmFree (CvMat*  matAr);
```

mat Pointer to the matrix.

Discussion

The function releases the memory allocated by the function `cvmAlloc`.

cvmFreeArray

Frees memory allocated for matrix array data.

```
void cvmFreeArray (CvMat*  matAr);
```

mat Pointer to the matrix array.

Discussion

The function releases the memory allocated by the function `cvmAllocArray`.

cvmAdd

Computes sum of two matrices.

```
void cvmAdd ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst);
```

<i>SrcA</i>	Pointer to the first source matrix.
<i>SrcB</i>	Pointer to the second source matrix.
<i>Dst</i>	Pointer to the destination matrix.

Discussion

This function adds the matrix *SrcA* to *Src2* and stores the result in *Dst*.

$$(c = a + b, c_i = a_i + b_i)$$

cvmSub

Computes difference sum of two matrices.

```
void cvmSum ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst );
```

<i>SrcA</i>	Pointer to the first source matrix.
<i>SrcB</i>	Pointer to the second source matrix.
<i>Dst</i>	Pointer to the destination matrix.

Discussion

The function subtracts the matrix *SrcB* from the matrix *SrcA* and stores the result in *Dst*.

$$(c = a - b, c_i = a_i - b_i)$$

cvmScale

Multiplies matrix by a value.

```
void cvmScale ( CvMat* Src, CvMat* Dst, Double value );
```

<i>Src</i>	Pointer to the source matrix.
------------	-------------------------------

Dst Pointer to the destination matrix.
value Factor.

Discussion

The function multiplies every element of the matrix by a value.

$$c = \alpha a, c_i = \alpha a_i$$

cvmDotProduct

Calculates dot product of two vectors in Euclidian metrics.

```
double cvmDotProduct(CvMat* Src1, CvMat* Src2);
```

Src1 Pointer to the first source vector.
Src2 Pointer to the second source vector.

Discussion

This function calculates the Euclidean dot product of two vectors and returns it.

$$DP = \sum_{i=1}^N a_i b_i$$

cvmCrossProduct

Calculates cross product of two 3D vectors.

```
void cvmCrossProduct( CvMat* Src1, CvMat* Src2, CvMat* Dest);
```

Src1 Pointer to the first source vector.
Src2 Pointer to the second source vector.

Dest Pointer to the destination vector.

Discussion

The function calculates the cross product of two vectors of length 3.

$$c = axb \quad (c_1 = a_2b_3 - a_3b_2, c_2 = a_3b_1 - a_1b_3, c_3 = a_1b_2 - a_2b_1)$$

cvmMul

Multiplies matrices.

```
cvmMul ( CvMat* SrcA, CvMat* SrcB, CvMat* Dst );
```

SrcA Pointer to the first source matrix.

SrcB Pointer to the second source matrix.

Dst Pointer to the destination matrix

Discussion

This function multiplies *SrcA* by *SrcB* and stores the result in *Dst*.

$$C = AB, C_{ij} = \sum_k A_{ik} B_{kj}$$

cvmMulTransposed

Calculates product of matrix and transposition.

```
void cvmMulTransposed (CvMat* Src, CvMat* Dst, Int order);
```

Src Pointer to the source matrix.

DestMatr Pointer to the destination matrix.

Order Order of multipliers.

Discussion

This function calculates the product of *SrcMatr* and its transposition.

The function evaluates $B = A^T A$ if *Order* = 0, $B = A A^T$ otherwise.

cvmTransp

Transposes the matrix.

```
ippStatus cvmTransp ( CvMat* Src, CvMat*Dst );
```

Src Pointer to the source matrix.

Dst Pointer to the destination matrix.

Discussion

This function transposes *Src* and stores result in *Dst*.

$$B = A^T, B_{ij} = A_{ji}$$

cvmInvert

Inverses matrix.

```
ippStatus cvmTransp ( CvMat* Src, CvMat*Dst );
```

Src Pointer to the source matrix.

Dst Pointer to the destination matrix.

Discussion

The function inverts *Src* and stores the result in *Dst*.

$$B = A^T, B_{ij} = A_{ji}$$

cvmTrace

Returns trace of matrix.

```
double cvmTrace ( CvMat* mat );
```

mat Pointer to the source matrix.

Discussion

The function returns the trace of the matrix *mat* .

cvmDet

Returns determinant of matrix.

```
double cvmDet ( CvMat* mat );
```

mat Pointer to the source matrix.

Discussion

The function returns the determinant of the matrix *mat*.

cvmCopy

Copies one matrix to another.

```
void cvmCopy ( CvMat* Src, CvMat* Dst );
```

Src Pointer to the source matrix.
Dest Pointer to the destination matrix.

Discussion

The function copies the matrix *Src* to the matrix *Dest*.

$$B = A, B_{ij} = A_{ij}$$

cvmSetZero_32f

Sets matrix to zero.

```
void cvmSetZero_32f ( CvMat* mat );
```

mat Pointer to the matrix to be set to zero.

Discussion

The function sets the matrix to zero.

$$A = 0, A_{ij} = 0$$

cvmSetIdentity

Sets matrix to identity.

```
void cvmSetIdentity ( CvMat* mat );
```

mat Pointer to the matrix to be set to identity.

Discussion

This function sets the matrix to identity.

$$A = E, A_{ij} = \delta_{ij}$$

cvmMahalonobis

Calculates Mahalonobis (weighted) distance between vectors.

```
double cvmMahalonobis ( CvMat* SrcA, CvMat* SrcB, CvMat* mat );
```

SrcA Pointer to the first source vector.

SrcB Pointer to the second source vector.

Matr Pointer to the weighted matrix.

Discussion

This calculates the weighted distance between two vectors and returns it.

$$Dist = \sqrt{\sum_i \sum_j T_{ij} (a_i - b_i)(a_j - b_j)}$$

cvmSVD

Provides single value decomposition.

```
void cvmSVD ( CvMat* Src, CvMat* Orth, CvMat* Diag );
```

Src Pointer to the source matrix

Orth Pointer to the matrix where the orthogonal matrix must be saved.

Diag Pointer to the matrix where the orthogonal matrix must be saved.

Discussion

The function represents the matrix *Src* as a product of the orthogonal matrix *Orth*, diagonal matrix *Diag*, and the matrix transposed to *Orth* correspondingly.



NOTE. The function `cvmSVD` destroys the source matrix `Src`. Therefore, in case the source matrix is needed after decomposition, the user is advised to clone it before running this function.

cvmEigenVV

Computes eigen values and eigen vectors.

```
void cvmEigenVV ( CvMat* Src, CvMat* evecs CvMat* evals, Double eps);
```

<i>Src</i>	Pointer to the source matrix.
<i>evecs</i>	Pointer to the matrix where eigenvectors must be stored.
<i>evals</i>	Pointer to the matrix where eigenvalues must be stored.
<i>eps</i>	Accuracy of diagonalization.

Discussion

The function computes the eigen values and eigen vectors of the matrix *Src* and stores them in the parameters *evals* and *evecs* correspondingly.



NOTE. The function `cvmEigenVV` destroys the source matrix `Src`. Therefore, if the source matrix is needed after eigenvalues calculation, the user is advised to clone it before running the function `cvmEigenVV`.

cvmPerspectiveProject

Implements general transform of 3D vector array.

```
void cvmPerspectiveProject (CvMat* mat, CvMatArray src, CvMatArray dst);
```

points 4x4 matrix.
src Source array of 3D vectors.
dst Destination array of 3D vectors.

Discussion

The function maps every input 3D vector $(x, y, z)^T$ to $(x'/w, y'/w, z'/w)^T$, where $(x', y', z', w')^T = (matr) \times (x, y, z, 1)^T$ and $w = \begin{cases} w', w' \neq 0 \\ 1, w' = 0 \end{cases}$.

This chapter describes functions that operate on eigen objects.

Reference

cvCalcCovarMatrixEx

Calculates covariance matrix for group of input objects.

```
void cvCalcCovarMatrixEx( int nObjects, void* input, int ioFlags, int
    ioBufSize, uchar* buffer, void* userData, IplImage* avg, float*
    covarMatrix );
```

<i>nObjects</i>	Number of source objects.
<i>input</i>	Pointer either to the array of <i>IplImage</i> input objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags (see Discussion).
<i>ioBufSize</i>	Input/output buffer size.
<i>buffer</i>	Pointer to input/output buffer.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>avg</i>	Averaged object.
<i>covarMatrix</i>	Covariance matrix. Output parameter, must be allocated before the call.

cvCalcEigenObjects

Calculates orthonormal eigen basis and mean object for group of input objects.

```
void cvCalcEigenObjects ( int nObjects, void* input, void* output, int ioFlags,
    int ioBufSize, void* userData, CvTermCriteria* calcLimit, IplImage* avg,
    float* eigVals;
```

<i>nObjects</i>	Number of source objects.
<i>input</i>	Pointer either to array of IplImage input objects or to read callback function (depending on <i>ioFlags</i>).
<i>output</i>	Pointer either to array eigen objects or to write callback function (depending on <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags (see Discussion).
<i>ioBufSize</i>	Input/output buffer size in bytes. The size is zero if unknown.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>calcLimit</i>	Determines the calculation finish conditions (see discussion).
<i>avg</i>	Averaged object.
<i>eigVals</i>	Pointer to the eigen values array in the descending order. May be NULL.

Discussion

Depending on the parameter *calcLimit*, calculations are finished either if the eigen faces number comes up to a certain value or if the relation between the current and the largest eigen values comes down to a certain value, or any of the above conditions takes place. The value *calcLimit->type* must be `IPPI_TERMCRIT_NUMB`, `IPPI_TERMCRIT_EPS`, or `IPPI_TERMCRIT_NUMB | IPPI_TERMCRIT_EPS`. The function returns the real values *calcLimit->maxIter* and *calcLimit->epsilon*.

The parameter *eigVals* may be equal to `NULL`, if eigen values are needless.

cvCalcDecompCoeff

Calculates decomposition coefficient of input object.

```
double cvCalcDecompCoeff( IplImage* obj, IplImage* eigObj, IplImage* avg );
```

<i>obj</i>	Input object.
<i>eigObj</i>	Eigen object.
<i>avg</i>	Averaged (mean) object.

Discussion

The function calculates the decomposition coefficient of the input object using the previously calculated eigen object and the mean (averaged) object.

cvEigenDecomposite

Calculates all decomposition coefficients for input object.

```
void cvEigenDecomposite( IplImage* obj, int nEigObjs, void* eigInput, int
    ioFlags, void* userData, IplImage* avg, float* coeffs );
```

<i>obj</i>	Input object.
<i>nEigObjs</i>	Number of eigen objects.
<i>eigInput</i>	Pointer either to the array of <i>IplImage</i> eigen objects or to the read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags (see Discussion).
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>avg</i>	Averaged object.

coeffs Calculated coefficients (output data).

Discussion

The function calculates all decomposition coefficients for the input object using the previously calculated eigen objects basis and the mean (averaged) object.

cvEigenProjection

Calculates object projection to the eigen sub-space (restores an object) using previously calculated eigen objects basis, mean (averaged) object and decomposition coefficients of the restored object.

```
void cvEigenProjection ( int nEigObjs, void* eigInput, int ioFlags, void*
    userData, float* coeffs, IplImage* avg, IplImage* proj );
```

<i>nEigObjs</i>	Number of eigen objects.
<i>eigInput</i>	Pointer either to array of IplImage eigen objects or to read callback function (depending on the parameter <i>ioFlags</i>).
<i>ioFlags</i>	Input/output flags (see Discussion).
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.
<i>coeffs</i>	Previously calculated decomposition coefficients.
<i>avg</i>	Averaged object.
<i>proj</i>	Decomposed object projection to the eigen sub-space.

Discussion

Terms and formulae

Let us define an object $u = \{u_1, u_2, \dots, u_n\}$ in the n -dimensional space as a sequence of values u_1 that could be vectors, images, etc. Images may either have or not have ROI. Let's assume that we have a group of input objects $u^i = \{u_1^i, u_2^i, \dots, u_n^i\}$, $i = 1, \dots, m$; usually $m \ll n$. Averaged, or mean, object $\bar{u} = \{\bar{u}_1, \bar{u}_2, \dots, \bar{u}_n\}$ of this group is defined as follows

$$\bar{u}_l = \frac{1}{m} \sum_{k=1}^m u_l^k$$

Covariance matrix $C = |c_{ij}|$ is a square symmetric matrix $m \times m$:

FORMULA.

Eigen objects basis $e^i = \{e_1^i, e_2^i, \dots, e_n^i\}$, $i = 1, \dots, m_1 \leq m$ of the input objects group may be calculated using the following relation:

$$e_l^i = \frac{1}{\sqrt{\lambda_i}} \sum_{k=1}^m v_k^i \cdot (u_l^k - \bar{u}_l),$$

where λ_i and $v^i = \{v_1^i, v_2^i, \dots, v_m^i\}$ are eigen values and the corresponding eigen vectors of the matrix C .

Any input object u^i as well as any other object u may be decomposed in the eigen objects m_1 -D sub-space. Decomposition coefficients of the object u are:

$$w_i = \sum_{l=1}^n e_l^i \cdot (u_l - \bar{u}_l)$$

Using these coefficients, we may calculate projection $\tilde{u} = \{\tilde{u}_1, \tilde{u}_2, \dots, \tilde{u}_n\}$ of the object u to the eigen objects sub-space, or, in other words, restore the object u in that sub-space):

$$\tilde{u}_l = \sum_{k=1}^{m_l} w_k e_l^k + \bar{u}_l$$

Use of functions

The functions of the eigen objects group have been developed to be used for any number of objects, even if their total size exceeds free RAM size. So the functions may be used in two main modes.

Direct access mode is the best choice if the size of free RAM is sufficient for all input and eigen objects allocation. This mode is set if the parameter *ioFlags* is equal to *CV_EIGOBJ_NO_CALLBACK*. In this case input and output parameters are pointers to arrays of input (output) objects addresses. The parameters *ioBufSize* and *userData* are not used. An example of the use of the function *cvCalcEigenObjects* use in direct access mode is shown below.

```
IplImage**  objects;
IplImage**  eigenObjects;
IplImage*   avg;
float*       eigVals;
CvSize       size = cvSize( nx, ny );
. . . . .
if( !( eigVals = (IplImage*) malloc( nObjects*sizeof(IplImage) ) ) )
    __ERROR_EXIT__;
if( !( avg = cvCreateImage( size, IPL_DEPTH_32F, 1 ) ) )
    __ERROR_EXIT__;
for( i=0; i< nObjects; i++ )
{
    objects[i]      = cvCreateImage( size, IPL_DEPTH_8U,  1 );
    eigenObjects[i] = cvCreateImage( size, IPL_DEPTH_32F, 1 );
    if( !( objects[i] & eigenObjects[i] ) )
        __ERROR_EXIT__;
}
```

```

. . . . .
cvCalcEigenObjects ( nObjects,
                    (void*)objects,
                    (void*)eigenObjects,
                    CV_EIGOBJ_NO_CALLBACK,
                    0,
                    NULL,
                    calcLimit,
                    avg,
                    eigVals );

```

The callback mode is the right choice in case when the number and the size of objects are large, which happens when all objects and/or eigen objects cannot be allocated in free RAM. In this case input/output information may be read/written and developed by portions. Such usage regime is called callback mode and is set by the parameter *ioFlags*. Three kinds of the callback mode may be set:

IoFlag = IPPI_EIGOBJ_INPUT_CALLBACK, only input objects are read by portions;

IoFlag = IPPI_EIGOBJ_OUTPUT_CALLBACK, only eigen objects are calculated and written by portions;

IoFlag = IPPI_EIGOBJ_BOTH_CALLBACK, or *IoFlag* = IPPI_EIGOBJ_INPUT_CALLBACK | IPPI_EIGOBJ_OUTPUT_CALLBACK, both processes take place. If one of the above modes is realized, the parameters *input* and *output*, both or either of them, are pointers to read / write callback functions. These functions must be written by the user; their prototypes are the same:

```
IppiStatus callback_read ( int ind, void* buffer, void* userData);
```

```
IppiStatus callback_write( int ind, void* buffer, void* userData);
```

<i>ind</i>	Index of the read or written object.
<i>buffer</i>	Pointer to the start memory address where the object will be allocated.
<i>userData</i>	Pointer to the structure that contains all necessary data for the callback functions.

The user must define the user data structure which may carry all information necessary to read/write procedure, such as the start address or file name of the first object on the HDD or any other device, row length and full object length, etc.

The function allocates a buffer in RAM for objects/eigen objects portion storage. The size of the buffer may be defined either by the user or automatically. If the parameter *ioBufSize* is equal to 0, or too large, the function will define the buffer size. The read data must be located in the buffer compactly, i.e., row after row, without alignment and gaps.

An example of the user data structure, i/o callback functions and the use of the function `ippiCalcEigenObjects_8u32fR` in the callback mode is shown below.

Example 24-1 User data structure, i/o callback functions and use of function `ippiCalcEigenObjects_8u32fR` in callback mode

```
// User data structure
typedef struct _UserData
{
    int      objLength; /* Obj. length (in elements, not in bytes !)
*/
    int      step;      /* Obj. step   (in elements, not in bytes !) */
    IppiSize size;      /* ROI or full size */
    IppiPoint roiIndent;
    char*    read_name;
    char*    write_name;
} UserData;
//-----
---
// Read callback function
IppiStatus callback_read_8u ( int ind, void* buffer, void* userData)
{
    int i, j, k = 0, m;
    UserData* data = (UserData*)userData;
    uchar* buff  = (uchar*)buf;
```

```

char name[32];
FILE *f;

if( ind<0 ) return IPP_BADFACTOR_ERR;
if( buf==NULL || userData==NULL ) IPP_NULLPTR_ERR;

for(i=0; i<28; i++)
{
    name[i] = data->read_name[i];
    if(name[i]=='.' || name[i]==' '))break;
}
name[i] = 48 + ind/100;
name[i+1] = 48 + (ind%100)/10;
name[i+2] = 48 + ind%10;
if((f=fopen(name, "r"))==NULL) return IPP_NOTDEFINED_ERR;
m = data->roiIndent.y*step + data->roiIndent.x;

for( i=0; i<data->size.height; i++, m+=data->step )
{
    fseek(f, m , SEEK_SET);
    for( j=0; j<data->size.width; j++, k++ )
        fread(buff+k, 1, 1, f);
}

fclose(f);
return 0;
}
//-----
---
// Write callback function
IppiStatus callback_write_32f ( int ind, void* buffer, void* userData)
{

```



```

    int i, j, k = 0, m;
    UserData* data = (UserData*)userData;
    float* buff = (float*)buf;
    char name[32];
    FILE *f;

    if( ind<0 ) return IPP_BADFACTOR_ERR;
    if( buf==NULL || userData==NULL ) IPP_NULLPTR_ERR;

    for(i=0; i<28; i++)
    {
        name[i] = data->read_name[i];
        if(name[i]=='.' || name[i]==' '))break;
    }
    if((f=fopen(name, "w"))==NULL) return IPP_NOTDEFINED_ERR;
    m = 4 * (ind*data->objLength + data->roiIndent.y*step
            + data->roiIndent.x);

    for( i=0; i<data->size.height; i++, m+=4*data->step )
    {
        fseek(f, m , SEEK_SET);
        for( j=0; j<data->size.width; j++, k++ )
            fwrite(buff+k, 4, 1, f);
    }

    fclose(f);
    return 0;
}
//-----
---
// fragments of the main function
{

```

```

. . . . .
    int bufSize = 32*1024*1024;  //32 MB RAM for i/o buffer
    float* avg;
    UserData data;
    IppiStatus r;
    IppiStatus (*read_callback)( int ind, void* buf, void* userData)=
        read_callback_8u;
    IppiStatus (*write_callback)( int ind, void* buf, void* userData)=
        write_callback_32f;
    IppiInput* u_r = (IppiInput*)&read_callback;
    IppiInput* u_w = (IppiInput*)&write_callback;
    void* read_    = (u_r)->data;
    void* write_   = (u_w)->data;
. . . . .
    data->read_name  = "input";
    data->write_name = "eigens";
    avg = (float*)_ipcvAlloc(sizeof(float) * obj_width * obj_height );

    cvCalcEigenObjects( obj_number,
                        read_,
                        write_,
                        IPPI_EIGOBJ_BOTH_CALLBACK,
                        bufSize,
                        (void*)&data,
                        &limit,
                        avg,
                        eigVal );
. . . . .
}

```


Embedded Hidden Markov Models

25

This chapter describes functions for using Embedded Hidden Markov Models (HMM) in face recognition task.

Overview

HMM structures

In order to support embedded models the user must define structures to represent 1D HMM and 2D embedded HMM model.

```
typedef struct _CvEHMM
{
    int level;
    int num_states;
    float* transP;
    float** obsProb;
    union
    {
        CvEHMMState* state;
        struct _CvEHMM* ehmm;
    } u;
}CvEHMM;
```

Below follows a list of parameters used for the definition of the structure

<i>level</i>	Level of embedded HMM. If <i>level</i> ==0, HMM is most external. (In 2D HMM there are two types of HMM: 1 external and several embedded. External HMM has <i>level</i> ==1, embedded HMMs have <i>level</i> ==0).
<i>num_states</i>	Number of states in 1D HMM
<i>transP</i>	State-to-state transition probability, square matrix (<i>num_state</i> × <i>num_state</i>).
<i>obsProb</i>	Observation probability matrix.
<i>state</i>	Array of HMM states. For the last-level HMM, i.e., an HMM without embedded HMMs, HMM states are “real”.
<i>ehmm</i>	Array of embedded HMMs. If HMM is not last-level, then HMM states are not “real” and they are HMMs.

For representation of observations the following structure is defined.

```
typedef struct CvImgObsInfo
{
    int obs_x;
    int obs_y;
    int obs_size;
    float** obs;
    int* state;
    int* mix;
}CvImgObsInfo;
```

This structure is used for storing observation vectors extracted from 2D image.

<i>obs_x</i>	Number of observations in the horizontal direction.
<i>obs_y</i>	Number of observations in the vertical direction.
<i>obs_size</i>	Length of every observation vector
<i>obs</i>	Pointer to observation vectors stored consequently. Number of vectors is <i>obs_x</i> * <i>obs_y</i> .
<i>state</i>	Array of indices of states, assigned to every observation vector
<i>mix</i>	Mixture assigned to the observation vector within an assigned state.

Reference

cvCreate2DHMM

Creates 2D embedded HMM.

```
void cvCreate2DHMM( CvEHMM** hmm, int* state_number, int* num_mix, int
                    obs_size );
```

<i>hmm</i>	Address of pointer to HMM to be created.
<i>state_number</i>	The first element of the parameter <i>state_element</i> defines the array that specifies number of states of external HMM, i.e., the number of embedded models; all subsequent elements define the number of states in every embedded HMM. Length of the array <i>state_number</i> is <i>state_number</i> [0]+1.
<i>num_mix</i>	Array with numbers of Gaussian mixtures.
<i>obs_size</i>	Size of observation vectors to be used with created HMM.

Discussion

The function creates a structure of the type `CvEHMM` with specified parameters.

cvRelease2DHMM

Releases 2D embedded HMM.

```
void cvRelease2DHMM( CvEHMM** hmm );
```

<i>hmm</i>	Address of pointer to HMM to be released.
------------	---

Discussion

The function frees all memory used by HMM and clears the pointer to HMM.

cvCreateObsInfo

Creates structure to store image observation vectors.

```
void cvCreateObsInfo( CvImgObsInfo** obs_info, CvSize num_obs, int obs_size );
```

<i>obs_info</i>	Address of pointer to the <code>CvImgObsInfo</code> structure to be created.
<i>num_obs</i>	Numbers of observations in the horizontal and vertical directions. For the given image and scheme of extracting observations the parameter can be computed via the macro <code>IPPI_COUNT_OBS(roi, dctSize, delta, num_obs)</code> , where <i>roi</i> , <i>dctSize</i> , <i>delta</i> , <i>num_obs</i> are the pointers to structures of the type <code>CvSize</code> .
<i>obs_size</i>	Size of observation vectors to be stored in the structure.

Discussion

The function creates new structures to store image observation vectors. For definitions of the parameters *roi*, *dctSize*, and *delta* see the specification of the function `cvImgToObs_DCT`.

cvReleaseObsInfo

Releases structure to store image observation vectors.

```
void cvReleaseObsInfo( CvImgObsInfo** obs_info );
```

<i>obs_info</i>	Address of the pointer to the structure <code>CvImgObsInfo</code> .
-----------------	---

Discussion

The function frees all memory used by observations and the clears pointer to the structure `CvImgObsInfo`.

cvImgToObs_DCT

Extracts observation vectors from image.

```
void cvImgToObs_DCT( IplImage* image, float* obs, CvSize dctSize, CvSize  
    obsSize, CvSize delta );
```

<i>image</i>	Input image.
<i>obs</i>	Pointer to consequently stored observation vectors.
<i>dctSize</i>	Size of image blocks for which DCT coefficients are to be computed.
<i>obsSize</i>	Number of the lowest DCT coefficients in the horizontal and vertical directions that will be put into the observation vector.
<i>delta</i>	Shift in pixels between two consecutive image blocks in the horizontal and vertical directions.

Discussion

The function extracts observation vectors (DCT coefficients) from the image. The user must pass *obs_info.obs* as the parameter *obs* to use this function with other HMM functions and use the structure *obs_info* of the type *CvImgObsInfo*.

Example 25-1

```
CvImgObsInfo* obs_info;  
.....  
ippiImgToObs_DCT( image,  
                  obs_info->obs, //!!!  
                  dctSize, obsSize, delta );
```

cvUniformImgSegm

Performs uniform segmentation of image observations by HMM states.

```
void cvUniformImgSegm( CvImgObsInfo* obs_info, CvEHMM* hmm);
```

obs_info Observations structure.

hmm HMM structure.

Discussion

The function segments image observations by HMM states uniformly (see picture for 2D embedded HMM with 5 superstates and 3, 6, 6, 6, 3 internal states of every corresponding superstate).



cvInitMixSegm

Segments all observations within every internal HMM state by state mixtures.

```
void cvInitMixSegm( CvImgObsInfo** obs_info_array, int num_img, CvEHMM* hmm);
```

obs_info_array Array of pointers to the observation structures.

num_img Length of above array.

hmm HMM.

Discussion

The function takes a group of observations from several training images already segmented by states and assigns a mixture to every observation vector.

cvEstimateHMMStateParams

Estimates all parameters of every HMM state.

```
void cvEstimateHMMStateParams(CvImgObsInfo** obs_info_array, int num_img,
                              CvEHMM* hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
<i>num_img</i>	Length of the array.
<i>hmm</i>	HMM.

Discussion

The function computes all inner parameters of every HMM state, including mixture means, variances, etc.

cvEstimateTransProb

Computes transition probability matrices for embedded HMM.

```
void cvEstimateTransProb( CvImgObsInfo** obs_info_array, int num_img, CvEHMM*
                          hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
<i>num_img</i>	Length of above array.
<i>hmm</i>	HMM.

Discussion

The function uses current segmentation of image observations to compute the number of transitions from one state to another.

cvEstimateObsProb

Computes probability of every observation of several images.

```
void cvEstimateObsProb( CvImgObsInfo* obs_info, CvEHMM* hmm);
```

obs_info Observation structure.

hmm HMM structure.

Discussion

The function computes Gaussian probabilities of each observation to occur in each of the internal HMM states.

cvEViterbi

Executes Viterbi algorithm for embedded HMM.

```
Float cvEViterbi( CvImgObsInfo* obs_info, CvEHMM* hmm);
```

obs_info Observation structure.

hmm HMM structure.

Discussion

Viterbi algorithm evaluates the likelihood of matching given image observations with a given HMM and performs segmentation of image observations by HMM states.

cvMixSegmL2

*Segments observations from all training images
by mixtures of newly assigned states.*

```
void cvMixSegmL2( CvImgObsInfo** obs_info_array, int num_img, CvEHMM* hmm);
```

<i>obs_info_array</i>	Array of pointers to the observation structures.
<i>num_img</i>	Length of the array.
<i>hmm</i>	HMM.

Discussion

The function segments observations from all training images by mixtures of newly Viterbi algorithm-assigned states. The function uses Euclidean distance to group vectors around existing mixtures.

Drawing Primitives

26

This chapter describes Drawing Primitives.

Reference

cvLine, cvLineAA

Draws simple, thick, or antialiased line segment.

```
void cvLine( IplImage* img, CvPoint pt1, CvPoint pt2, int thickness, int color
);
```

```
void cvLineAA( IplImage* img, CvPoint pt1, CvPoint pt2, int scale, int color );
```

<i>img</i>	Image.
<i>pt1</i>	First point of the line segment.
<i>pt2</i>	Second point of the line segment.
<i>thickness</i>	Line thickness.
<i>scale</i>	Number of fractional bits in the end point coordinates.
<i>color</i>	Line color (RGB) or brightness (grayscale image).

Discussion

The functions draw the line segment between *pt1* and *pt2* points on the image. The line is clipped by the image or ROI rectangle. The Bresenham algorithm is used for simple line segments. Thick lines are drawn with rounding endings, the antialiased line

drawing algorithm includes some sort of Gaussian filtering to get smooth picture. To specify the line color the user may use the macro `CV_RGB (r, g, b)` that makes a 32-bit color value from the color components.

cvRectangle

Draws simple or thick rectangle.

```
void cvRectangle( IplImage* img, CvPoint pt1, CvPoint pt2, int thickness, int color );
```

<i>img</i>	Image.
<i>pt1</i>	One of the rectangle vertices.
<i>pt2</i>	Opposite rectangle vertex.
<i>thickness</i>	Thickness of lines that make up the rectangle.
<i>color</i>	Line color (RGB) or brightness (grayscale image).

Discussion

Draws rectangle with two opposite corners *pt1* and *pt2*. If the parameter *thickness* is positive or zero, the outline of the rectangle is drawn with that thickness, otherwise a filled rectangle is drawn.

cvCircle

Draws simple or filled circle.

```
void cvCircle( IplImage* img, CvPoint center, int radius, int color, int do_fill );
```

<i>img</i>	Image where the line is drawn.
<i>center</i>	Center of the circle.
<i>radius</i>	Radius of the circle.

color Circle color (RGB) or brightness (grayscale image).
do_fill The function fills the circle if the parameter is non-zero.

Discussion

The function draws the simple or filled circle with given center and radius. The circle is clipped by ROI rectangle. The Bresenham algorithm is used both for simple and filled circles. To specify the circle color the user may use the macro `CV_RGB (r, g, b)` that makes a 32-bit color value from the color components.

cvEllipse, cvEllipseAA

Draws a simple, thick, or antialiased elliptic arc.

```
void cvEllipse( IplImage* img, CvPoint center, CvSize axes, double angle,
               double start_angle, double end_angle, int thickness, int color );
void cvEllipseAA( IplImage* img, CvPoint center, CvSize axes, double angle,
                 double start_angle, double end_angle, int scale, int color );
```

img Image.
center Center of the ellipse.
axes Length of ellipse axes.
angle Rotation angle.
start_angle Starting angle of the elliptic arc.
end_angle Ending angle of the elliptic arc.
thickness Thickness of the the ellipse arc.
scale Specifies the number of fractional bits in the center coordinates and axes sizes.
color Ellipse color (RGB) or brightness (grayscale image).

Discussion

The functions draw the simple, thick or antialiased elliptic arc. The arc is clipped by ROI rectangle. Generalized Bresenham algorithm for conic section is used for simple elliptic arcs here, and piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are in degrees. The meaning of parameters is shown on the picture below:

cvFillPoly, cvFillConvexPoly

Fills a convex or arbitrary polygon.

```
void cvFillPoly( IplImage* img, int contours, CvPoint** pts, int* npts, int
                color );
```

```
void cvFillConvexPoly( IplImage* img, CvPoint* pts, int npts, int color );
```

<i>img</i>	Image.
<i>contours</i>	Number of contours that bind the filled region.
<i>pts</i>	Array of pointers to polygons (cvFillPoly case) or a single polygon (cvConvexFillPoly case).
<i>npts</i>	Array of array counters or a single counter.
<i>color</i>	Polygon color (RGB) or brightness (grayscale image).

Discussion

The functions fill the convex or arbitrary polygon with the specified color. The function `cvFillConvexPoly` is much faster and fills not only the convex polygon but any monotonic polygon, i.e., a polygon, whose contour intersects every horizontal line (scan line) twice at the most. The function `cvFillPoly` fills more complex areas, e.g., areas with holes, contour self-intersection etc.

cvPolyLine, cvPolyLineAA

Draws simple, thick, or antialiased polyline.

```
void cvPolyLine( IplImage* img, int contours, CvPoint** pts, int* npts,
                 is_closed, int thickness, int color );
void cvPolyLineAA( IplImage* img, int contours, CvPoint** pts, int* npts,
                  is_closed, int scale, int color );
```

<i>img</i>	Image.
<i>contours</i>	Number of polyline contours.
<i>pts</i>	Array of pointers to polylines.
<i>npts</i>	Array of polyline counters or a single counter.
<i>is_closed</i>	Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.
<i>thickness</i>	Thickness of the polyline edges.
<i>scale</i>	Specifies number of fractional bits in the center coordinates and axes sizes.
<i>color</i>	Polygon color (RGB) or brightness (grayscale image).

Discussion

The functions draw a set of simple, thick, or antialiased polylines.

cvInitFont

Initializes font structure.

```
void cvInitFont( CvFont* font, CvFontFace font_face, float hscale, float
                 vscale, float italic_scale, int thickness );
```

<i>font</i>	Pointer to the resultant font structure.
-------------	--

<i>font_face</i>	Font name identifier. Only the font <code>CV_FONT_VECTOR0</code> is currently supported.
<i>hscale</i>	Horizontal scale. If equal to <code>1.0f</code> , the characters will have the original width depending on the font type. If equal to <code>0.5f</code> , the characters will be half of the original width.
<i>vscale</i>	Vertical scale. If equal to <code>1.0f</code> , the characters will have the original height depending on the font type. If equal to <code>0.5f</code> , the characters will be half of the original height.
<i>italic_scale</i>	Approximate tangent of the character slope relative to the vertical line. Zero value means a non-italic font, <code>1.0f</code> means $\sim 45^\circ$ slope, etc.
<i>thickness</i>	Thickness of lines composing letters outlines. The function <code>cvLine</code> is used for drawing letters.

Discussion

The function initializes the font structure that can be passed further into text drawing functions. Although only one font is supported, it is possible to get different font “flavors” by varying the scale parameters, slope, and thickness.

cvPutText

Draws the text string.

```
void cvPutText( IplImage* img, const char* text, CvPoint org, CvFont* font, int
                color );
```

<i>img</i>	Input image.
<i>text</i>	String to print.
<i>org</i>	Coordinates of bottom-left corner of the first letter.
<i>font</i>	Pointer to the font structure.
<i>color</i>	Text color (RGB) or brightness (grayscale image).

Discussion

The function draws the text on the image with the specified font and color. The printed text is clipped by ROI rectangle. Symbols that do not belong to the specified font are replaced with the “rectangle” symbol.

cvGetTextSize

Retrieves width and height of text string.

```
void cvGetTextSize( CvFont* font, const char* text_string, CvSize* text_size,
    int* ymin );
```

<i>font</i>	Pointer to the font structure.
<i>text_string</i>	Input string.
<i>text_size</i>	Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.
<i>ymin</i>	Lowest <i>y</i> coordinate of the text relative to the baseline. Negative, if the text includes such characters as g, j, p, q, y etc. and zero otherwise.

Discussion

The function calculates the binding rectangle for the given text string when the specified font is used.

System Functions

27

This chapter describes system library functions.

Reference

cvLoadPrimitives

Loads optimized versions of functions for specific platform.

```
int cvLoadPrimitives (char* dllName, char* processor_type);
```

<i>dll</i>	Name of dll without postfix that contains the optimized versions of functions
<i>processor_type</i>	Postfix that specifies the platform type: “w7” for Willamette, “A6” for Intel Pentium III, “M6” for Intel Pentium II, NULL for auto detection of the platform type.

Discussion

The function loads the versions of functions that are optimized for a specific platform. The function is automatically called before the first call to the library function, if not called earlier.

cvGetLibraryInfo

Gets the library information string.

```
void cvGetLibraryInfo (char** version, int* loaded, char** dll_name);
```

version Pointer to the string that will receive the build date information (can be NULL).

loaded Postfix that specifies the platform type:
 “W7” for Willamette, “A6” for Intel Pentium III, “M6” for Intel Pentium II, NULL for auto detection of the platform type.

dll_name Pointer to the full name of dll without path, could be NULL.

Discussion

TBD

This chapter describes image statistics functions.

Reference

cvAbsDiff, cvAbsDiffS

Calculates absolute difference between two images and between image and scalar value.

```
void cvAbsDiff( IplImage* srcA, IplImage* srcB, IplImage* dst );  
void cvAbsDiff( IplImage* srcA, IplImage* dst, double value );
```

<i>srcA</i>	First compared image.
<i>srcB</i>	Second compared image.
<i>dst</i>	Destination image.
<i>value</i>	Value to compare.

Discussion

The functions calculate the absolute difference between two images or between an image and a scalar value.

```
cvAbsDiff:  $dst[i] = abs(src[i] - dst[i]);$   
cvAbsDiffS:  $dst[i] = abs(src[i] - value).$ 
```

cvMatchTemplate

Fills characteristic image for given image and template.

```
void cvMatchTemplate( IplImage* img, IplImage* templ, IplImage* result,
    CvTemplMatchMethod method );
```

<i>img</i>	Image where the search is running.
<i>templ</i>	Searched template. Must be not greater than the parameter <i>img</i> .
<i>result</i>	Output characteristic image. If the parameter <i>img</i> has the size of $W \times H$ and the template has the size of size $w \times h$, the resulting image must have the size or selected ROI $W - w + 1 \times H - h + 1$.
<i>method</i>	Specifies the way the template must be compared with image regions.

Discussion

The function implements a set of methods for finding regions on the image that are similar to the given template.

Given a source image with $W \times H$ pixels and template with $w \times h$ pixels, we get the resulting image with $W - w + 1 \times H - h + 1$ pixels, and the pixel value in each location (x, y) characterizes the similarity between the template and the image rectangle with the top-left corner at (x, y) and the right-bottom corner at $(x + w - 1, y + h - 1)$. Similarity can be calculated in several ways:

Squared difference (method == CV_TM_SQDIFF)

$$S(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} [T(x', y') - I(x + x', y + y')]^2,$$

where $I(x, y)$ is the value of the image pixel in the location (x, y) , $T(x, y)$ is the value of the template pixel in the location (x, y) .

Normalized squared difference (method == CV_TM_SQDIFF_NORMED)

$$S(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} [T(x', y') - I(x+x', y+y')]^2}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} I(x+x', y+y')^2}}$$

Cross correlation (method == CV_TM_CCORR):

$$C(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y') I(x+x', y+y')$$

Cross correlation, normalized (method == CV_TM_CCORR_NORMED):

$$\tilde{C}(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y') I(x+x', y+y')}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} T(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} I(x+x', y+y')^2}}$$

Correlation coefficient (method == CV_TM_CCOEFF):

$$R(x, y) = \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y') \tilde{I}(x+x', y+y')$$

where $\tilde{T}(x', y') = T(x', y') - \bar{T}$, $\tilde{I}(x+x', y+y') = I(x+x', y+y') - \bar{I}(x, y)$, and where \bar{T} stands for the average value of pixels in the template raster and $\bar{I}(x, y)$ stands for the average value of the pixels in the current “window” of the image.

Correlation coefficient, normalized (method == CV_TM_CCOEFF_NORMED):

$$\tilde{R}(x, y) = \frac{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y') \tilde{I}(x+x', y+y')}{\sqrt{\sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{T}(x', y')^2 \sum_{y'=0}^{h-1} \sum_{x'=0}^{w-1} \tilde{I}(x+x', y+y')^2}}$$

After the function returns the resultant image, probable positions of the template in the image could be located as the local or global maximums of the resultant image brightness.

cvCvtPixToPlane

Divides pixel image into separate planes.

```
void cvCvtPixToPlane( IplImage* src, IplImage* dst0, IplImage* dst1, IplImage*
dst2, IplImage* dst3);
```

src Source image.
dst0...dst4 Destination planes.

Discussion

The function divides a color image into separate planes. Two modes are available for the operation. Under the first mode the parameters *dst0*, *dst1*, and *dst2* are non-zero for the three-channel source image (*dst3* must be zero if the source has three channels); for the four-channel source image all the destination image pointers are nonzero, in this case the function splits the three/four channel image into separate planes and writes them to destination images. Under the second mode only one of the destination images is not `NULL`; in this case, the corresponding plane is extracted from the image and placed into destination image.

cvCvtPlaneToPix

Composes color image from separate planes.

```
void cvCvtPlaneToPix( IplImage* src0, IplImage* src1, IplImage* src2,  
    IplImage* src3, IplImage* dst );
```

src0...src4 Source planes.

dst Destination image.

Discussion

The function composes color image from separate planes. If the *dst* has three channels, then *src0*, *src1*, and *src2* must be nonzero, otherwise *dst* must have four channels and all the source images must be non-zero.

cvConvertScale

Converts one image to another with linear transformation.

```
void cvConvertScale( IplImage* src, IplImage* dst, double scale, double  
    shift );
```

src Source image.

dst Destination image.

Discussion

The function applies linear transform to all pixel in the source image and puts the result into the destination image with appropriate type conversion. Only two variants are currently available: conversion of unsigned char to float and float to unsigned char. The unsigned char to float conversion is effected by the formula

$$dst(x,y) = (float)(src(x,y)*scale + shift);$$

The float is converted to unsigned char by the the following algorithm

```
t = round(src(x,y)*scale + shift);
if( t < 0 )
dst(x,y) = 0;
else if( t > 255 )
dst(x,y) = 255;
else
dst(x,y) = (unsigned char)t;
```

cvInitLineIterator

Initializes line iterator.

```
int cvInitLineIterator( IplImage* img, CvPoint pt1, CvPoint pt2,
CvLineIterator* iterator);
```

<i>img</i>	Image.
<i>pt1</i>	Starting line point.
<i>pt2</i>	Ending line point.
<i>iterator</i>	Pointer to the line iterator state structure.

Discussion

The function initializes the line iterator and returns the number of pixels between two ending points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point. The points on the line are calculated one by one using the 8-point connected Bresenham algorithm. Below follows an example how to draw the line on the RGB image, such that the image pixels that belong to the line are mixed with the given color using the XOR operation.

```
void put_xor_line( IplImage* img, CvPoint pt1, CvPoint pt2, int r, int
g, int b ) {
CvLineIterator iterator;
```

```
int count = cvInitLineIterator( img, pt1, pt2, &iterator);
for( int i = 0; i < count; i++ ){
    iterator.ptr[0] ^= (uchar)b;
    iterator.ptr[1] ^= (uchar)g;
    iterator.ptr[2] ^= (uchar)r;
    CV_NEXT_LINE_POINT(iterator);
}
}
```

cvSampleLine

Reads raster line to buffer.

```
int cvSampleLine( IplImage* img, CvPoint pt1, CvPoint pt2, void* buffer );
```

<i>img</i>	Image.
<i>pt1</i>	Starting line point.
<i>pt2</i>	Ending line point.
<i>buffer</i>	Buffer to store the line points. Must have enough size to store $MAX(pt2.x - pt1.x + 1, pt2.y - pt1.y + 1)$ points.

Discussion

The function implements one particular case of application of line iterators. The function reads all the image points, lying on the line between *pt1* and *pt2*, including the ending points, and stores them to the buffer.

cvGetRectSubPix

Retrieves raster rectangle from image with sub-pixel accuracy.

```
void cvGetRectSubPix( IplImage* src, IplImage* rect, CvPoint2D32f center );
```

<i>src</i>	Source image.
<i>rect</i>	Extracted rectangle. Must have odd width and height.
<i>center</i>	Floating point coordinates of the rectangle center. The center must be inside the image.
<i>buffer</i>	Buffer to store the line points. Must be enough size to store $MAX(pt2.x - pt1.x + 1, pt2.y - pt1.y + 1)$ points.

Discussion

The function extracts pixels from *src*, if the pixel coordinates satisfy the conditions below:

```
center.x - (widthrect-1)/2 <= x <= center.x + (widthrect-1)/2;
center.y - (heightrect-1)/2 <= y <= center.y + (heightrect-1)/2
```

Since the center coordinates are not integer, bilinear interpolation is applied to get the values of pixels in non-integer locations. Although the rectangle center must be inside the image, the whole rectangle may be partially occluded. In this case, the pixel values are spread from the boundaries outside the image to approximate values of occluded pixels.

cvbFastArctan

Calculates fast arctangent approximation for arrays of abscissas and ordinates.

```
void cvbFastArctan( const float* y, const float* x, float* angle, int len );
```

<i>y</i>	Array of ordinates.
<i>x</i>	Array of abscissas.
<i>angle</i>	Calculated angles of points (<i>x</i> [<i>i</i>], <i>y</i> [<i>i</i>]).
<i>len</i>	Number of elements in the arrays.

Discussion

The function calculates an approximate arctangent value, the angle of the point (*x*,*y*). The angle is in the range 0°... 360°. Accuracy is about 0.1°. For point (0,0) the resultant angle is 0.

cvSqrt, cvbSqrt

Calculate square root of single float argument or array of floats.

```
float cvSqrt( float x );  
void cvbSqrt( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

These functions calculate the square root of their arguments. Arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than 9e-6, for the vector function less than 3e-7.

cvInvSqrt, cvbInvSqrt

Calculate inverse square root of single float argument or array of floats.

```
float cvInvSqrt( float x );  
void cvbInvSqrt( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

These functions calculate the inverse square root of their arguments. Arguments should be non-negative, otherwise the result is unpredictable. The relative error for the scalar version is less than $9e-6$, for the vector function less than $3e-7$.

cvbReciprocal

Calculates inverse of array of floats.

```
void cvbRecip( const float* x, float* y, int len );
```

<i>x</i>	Argument, scalar or array.
<i>y</i>	Resultant array.
<i>len</i>	Number of elements in the arrays.

Discussion

These functions calculate the inverse ($1/x$) of their arguments. Arguments should be non-zero. The functions give a very precise result with the relative error is less than $1e-7$.

cvbCartToPolar

Calculates magnitude and angle for array of abscissas and ordinates.

```
void cvbCartToPolar( const float* y, const float* x, float* mag, float* angle,
                    int len );
```

<i>y</i>	Array of ordinates.
<i>x</i>	Array of abscissas.
<i>mag</i>	Calculated magnitudes of points ($x[i]$, $y[i]$).
<i>angle</i>	Calculated angles of points ($x[i]$, $y[i]$).
<i>len</i>	Number of elements in the arrays.

Discussion

The function calculates the magnitude $\sqrt{x[i]^2 + y[i]^2}$ and the angle $\arctan(y[i]/x[i])$ of each point ($x[i]$, $y[i]$). The angle is measured in degrees and varies from 0° to 360°. The function is a combination of the functions `cvbFastArctan` and `cvbSqrt`, so the accuracy is the same as in these functions. If pointers to the angle array or the magnitude array are `NULL`, the corresponding part is not calculated.

cvbFastExp

Calculates fast exponent approximation for array of floats.

```
void cvbFastExp( const float* x, double* exp_x, int len);
```

<i>x</i>	Array of arguments.
<i>exp_x</i>	Array of results.
<i>len</i>	Number of elements in the arrays.

Discussion

The function calculates fast exponent approximation for each element of the input array. Maximal relative error is about $7e-6$.

cvbFastLog

Calculates fast approximation of natural logarithm for array of doubles.

```
void cvbFastLog( const double* x, float* log_x, int len);
```

<i>x</i>	Array of arguments.
<i>exp_x</i>	Array of results.
<i>len</i>	Number of elements in the arrays.

Discussion

The function calculates fast logarithm approximation for each element of the input array. Maximal relative error is about $7e-6$.

cvRandInit

Initializes state of random number generator.

```
void cvRandInit( CvRandState* state, float lower, float upper, int seed );
```

<i>state</i>	Pointer to the initialized random number generator state.
<i>lower</i>	Lower boundary of uniform distribution.
<i>upper</i>	Upper boundary of uniform distribution.
<i>seed</i>	Initial 32-bit value to start a random sequence.

Discussion

The function initializes the `state` structure that is used for generating uniformly distributed numbers in the range `[lower, upper)`. A multiply-with-carry generator is used.

cvbRand

Fills array with random numbers

```
void cvbRand( CvRandState* state, float* x, int len );
```

<i>state</i>	Random number generator state.
<i>x</i>	Destination array.
<i>len</i>	Number of elements in the array.

Discussion

The function fills the array with random numbers and updates generator state.

cvFillImage

Fills image with constant value.

```
void cvFillImage( IplImage* img, double val );
```

<i>img</i>	Filled image.
<i>val</i>	Value to fill the image.

Discussion

The function is equivalent to either `iplSetFP` or `iplSet`, depending on the pixel type (floating-point or integer).

cvRandSetRange

Sets range of generated random numbers without reinitializing RNG state.

```
void cvRandSetRange( CvRandState* state, double lower, double upper );
```

<i>state</i>	State of random number generator (RNG).
<i>lower</i>	New lower bound of generated numbers.
<i>upper</i>	New upper bound of generated numbers.

Discussion

The function changes the range of generated random numbers without reinitializing RNG state. For the current implementation of RNG the function is equivalent to the following code:

```
unsigned seed = state.seed;
unsigned carry = state.carry;
cvRandInit( &state, lower, upper, 0 );
state.seed = seed;
state.carry = carry;
```

However, the function is preferable because of compatibility with the next versions of the library.

cvKMeans

Adjusts mean vectors of every cluster.

```
void cvKMeansans, ( int num_clusters, CvVect32f* samples, int num_samples, int
vec_size, CvTermCriteria termcrit, int* cluster );
```

<i>num_clusters</i>	number of required clusters
<i>samples</i>	pointer to array of input vectors

<i>num_samples</i>	Number of input vectors.
<i>vec_size</i>	Size of every input vector.
<i>termcrit</i>	Criteria of iterative algorithm termination.
<i>cluster</i>	Characteristic array of cluster numbers, corresponding to each input vector.

Discussion

The function iteratively adjusts mean vectors of every cluster. Termination criteria must be used to stop the execution of the algorithm. At every iteration the convergence value is computed:

$$\sum_{i=1}^K \|old_mean_i - new_mean_i\|^2$$

The function terminates if $E < Termcrit.epsilon$.

This bibliography provides a list of publications that might be useful to the Intel® Computer Vision Library users. This list is not complete; it serves only as a starting point.

- [Fitzgibbon95] Andrew W. Fitzgibbon, R.B.Fisher. *A Buyer's Guide to Conic Fitting*, Proc.5th British Machine Vision Conference, Birmingham, pp.513-522, 1995.
- [Kass88] M. Kass, A. Witkin, and D. Terzopoulos. *Snakes: Active Contour Models*, International Journal of Computer Vision, pp321-331, 1988.
- [Matas] J.Matas, C.Galambos, J.Kittler. *Progressive Probabilistic Hough Transform*. British Machine Vision Conference.
- [Trucco98] Emanuele Trucco, Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, Inc., 1998.
- [Williams92] D. J. Williams and M. Shah. *A Fast Algorithm for Active Contours and Curvature Estimation*. CVGIP: Image Understanding, Vol. 55, No. 1, pp. 14-26, Jan.,1992._
<http://www.cs.ucf.edu/~vision/papers/shah/92/WIS92A.pdf>.
- [Yuille89] A.Y.Yuille, D.S.Cohen, and P.W.Hallinan. *Feature extraction from faces using deformable templates in CVPR*, pp.104-109, 1989.
- [Zhang96] Zhengyou Zhang. *Parameter Estimation Techniques: A Tutorial with Application to Conic Fitting*, to appear in Image and Vision Computing Journal, 1996.

Index

C

cvLaplace, 8

CV_MOVE_PARAM_WRAP, 12
CV_MOVE_TO, 10
CV_MOVE_WRAP, 11

I

Image Function Reference, 1
 cvCopyImage, 7
 cvCreateImage, 2
 cvCreateImageData, 3
 cvCreateImageHeader, 1
 cvGetImageRawData, 6
 cvInitImageHeader, 6
 cvReleaseImage, 3
 cvReleaseImageData, 4
 cvReleaseImageHeader, 2
 cvSetImageCOI, 5
 cvSetImageData, 4
 cvSetImageROI, 5

S

Sequence Function Reference, 7

M

Memory Storage Function Reference, 1
 cvClearMemStorage, 5
 cvCreateChildMemStorage, 4
 cvCreateMemStorage, 3
 cvRestoreMemStoragePos, 5
 cvSaveMemStoragePos, 5

P

Pixel Access Macro Reference, 7
 CV_MOVE, 11
 CV_MOVE_PARAM, 12